Organisation-based co-ordination of wireless sensor networks

by

María del Carmen Delgado Román

A dissertation presented in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy in Computer Science

Tutor:	*Supervisor:*	*PhD Candidate:*
Dr. Jordi González Sabaté	Dr. Carles Sierra García	María del Carmen Delgado Román

Universitat Autònoma
de Barcelona

October 6, 2014
Universitat Autònoma de Barcelona
Departament de Ciències de la Computació

Contents

List of Figures

Acknowledgements

It has been a long way until reaching the moment of finishing this thesis. Now, it is time to look back and feel grateful for all I have gained during these years, knowledge, skills, trips and the most important thing of all, the love and support of all of you who have been by my side.

First of all, I would like to thank the IIIA for funding my research through the Agreement Technologies project (funded by CONSOLIDER CSD 2007-0022, INGENIO 2010). Great thanks to all the IIIA members for being like a family and transmitting this sensation to everyday work. You have really created a nice environment where to work.

Special thanks to my supervisor, Carles Sierra, who has always encouraged me to do my best and get this work done. Thank you for all your help and patience when I needed it the most.

I have been very lucky of sharing my time in the lab with incredible fellows and friends! Thanks for the laughs, coffees, beers, advises, discussions...Thank you for sharing the stressing days before deadlines and for initiating me in the art of ping-pong playing! Many thanks to Marc, who has always listened to me and smartly advised me with my code problems.

To my friends, who are always by my side at the crucial moments of my life, either to laugh or to cry. Ale, Meritxell, Fabi and Montse, I cherish your friendship.

I have always felt grateful for the family in which I was born, now I am also blessed with a warm-hearted sensible husband and his family. Santi, thanks for believing in me and making my life cheerful. Thank you also for contributing, together with my mom, to make all my lab mates recognise my handy tune!

All my love and acknowledgement go to my mom, who made me the person I am today. It is completely impossible to express with words how much I admire her for all that she has taught and given to me, unconditionally, always. Her strength and courage led me to finish this work. This is thanks to her, and for her and my granny, strong women of my family. I miss you, and I will be missing you every single second of the rest of my life.

Thank you to all of you who have been by my side at any time along these years, just by that, you have contributed to this work and its completion, and I appreciate it. Thank you.

–M del Carmen Delgado Román

Chapter 1

Introduction

This chapter motivates the research problem studied and introduces a particular use case. The proposal developed to tackle the problem is outlined and the main contributions derived from this work are presented.

1.1 Motivation

Wireless Sensor Networks (WSNs) are networks formed by a large number of battery-operated sensing nodes able to develop monitoring tasks in different environments. Advances in Electronics and Telecommunication have favoured the development of small sensors able to communicate wirelessly and to perform different functions in the environments where they are deployed. Each node is a low-cost, low-consumption device of limited capabilities, yet capable of sensing its environment and communicating wirelessly. This fact, together with its decreasing price, have contributed to the growing use of WSNs [Akyildiz et al., 2002]. This technology allows to perform surveillance tasks in large physical spaces. Moreover, the large numbers of nodes make these networks very robust to individual node failures, enabling them to operate in remote and hazardous environments. These characteristics, together with their non-invasive nature, make WSNs appropriate for a great range of monitoring applications.

The change of perspective that WSNs introduced with respect to classical monitoring, together with their capacity for monitoring difficult-to-access environments, have attracted the attention of researchers from different areas.WSNs application domains are vast and continue getting more interest thanks to concepts such as smart city, smart metering or eHealth. The creation of intelligent environments that incorporate new functionalities, and allow for a more efficient performance, are essential goals of these techniques.

One of the first incursions of sensor network technology in daily life took place through domotics. A typical capability of intelligent buildings is to monitor the temperature of the room and to turn on/off the air conditioning in order to maintain the place at a target temperature. Nowadays, the functionalities

provided by this technology has significantly increased and allow for interaction with the house's tenants.

The development of smart cities relies on the creation of interconnected sensor networks across the cities. WSNs play a fundamental role in the definition and deployment of these networks that sample information about multiple aspects of the city, such as air pollution, noise level, street lighting, etcetera. The collection and adequate processing of these data provide with more efficient management of the city. Moreover, it also endows the city with added value functionalities that ease the life of its residents and visitants. Projects developed within this area are Smartcity Málaga [Endesa, 2014] and SmartSantander [Telefónica et al., 2014]. Among the different functionalities that they provide, we can cite an efficient management of public lighting in Málaga, the smart parking application in Santander or the SmartSantander augmented reality application, which may result of special interest for tourists.

These circumstances make WSNs an interesting research area, not only by themselves, but also for their applications' management. The growing number of monitoring applications leads to the generation of a large amount of data that has to be stored and processed in order to extract information from the surveyed areas. Store and processing of these data consume an important amount of energy. Consequently, it also represents a pollution source causing carbon dioxide emission. Therefore, an efficient management of WSNs that makes them sample the environment effectively poses as a challenging task.

Moreover, growing application of WSNs have entailed the proliferation of studies to alleviate the hard constraints to which nodes' devices are subject, either due to their physical characteristics or the features of the environments where they actuate. Sensor nodes are constraint in terms of computation, communication capacity and energy availability. These devices are typically battery-operated and can be deployed in difficult-to-access environments, therefore, preventing from battery replenishment. This situation has led to the development of numerous algorithms that pursue the extension of the network lifetime. The importance of extending the lifespan of the network is critical, as it provides the system with longer operation capacity. Despite this recognised cruciality, strategies developed to achieve this aim must also maintain a correct network operation performance, as not every strategy is adequate for every application scenario.

The energy management of a WSN is a key issue that affects the network from its design phase to its operation. Energy harvesting and energy conservation methods are considered elemental techniques to help in improving WSNs energy conditions. There are different approaches to energy conservation. Many contributions have focused in the design of routing techniques that reduce the number of transmissions that each node performs, thereafter, saving overall network energy.

The approach proposed in this thesis to tackle this problem focuses on **reducing the number of sampling actions that an agent takes to provide valid information to the sink**. Although the most costly action of a node

is usually transmission, research in Electronics has resulted in the development of more powerful sensors with growing energy demands. Decreasing sensing actuation avoids the energy consumption derived from this element actuation and the subsequent transmission action. Moreover, this strategy tries to eliminate redundant data generation, which also contributes to lower the energy demands for monitoring data storage and processing. Therefore, considering this sensor function poses a challenging problem that may provide interesting results to the WSNs community.

1.2 Use case

As it has already been introduced in the previous section, there is a huge range of possible applications for WSNs. Analogously as they are being deployed in cities, WSNs are also being applied to water environments. However, this kind of environments pose additional challenges due to their particular characteristics that demand robust nodes capable of working in hard environments.

Monitoring the state of the water in seas, lakes and waterways can derive important information about natural phenomena behaviour. Waterway surveillance can also detect changes in the water composition that warn about pollutant leakages or other phenomena altering regular conditions. The introduction of WSNs technology to this specific kind of environments is known as smart water. Some applications situated within this framework are monitoring the potable conditions of water, real-time control of leakages in the sea or river floods prediction.

An aquatic environment that may be especially attractive for a WSN deployment is a navigable river, a waterway. This kind of rivers presents a signalling infrastructure to guide the vessels navigation along their course. Frequently, they represent important routes for goods transportation and depending on the area, sightseeing boats also navigate along the waterway. Examples of rivers verifying these conditions are Ebro in northeast Spain and Guadalquivir in south Spain. Figure 1.1 shows a view of the mouth and course of Guadalquivir river.

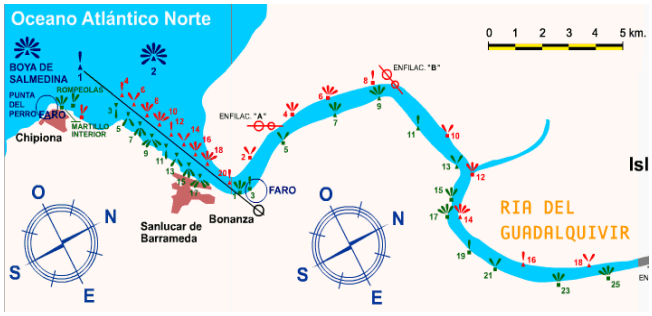


Figure 1.1: Mouth and course of Guadalquivir river.

The enhancement of the signalling infrastructure by the deployment of a

WSN provides management capabilities and added value functionalities to this environment. Activities that can benefit from this improvement are, for instance buoys lighting self-management or monitoring of the river level and water conditions in areas situated near irrigation fields. Nonetheless, developing a WSN in this water environment poses a challenging problem. Ambient conditions are harder than in urban areas. Therefore, devices must be durable. Moreover, nodes accessibility is constrained, and there is no possibility of nodes' connection to the power net. In these circumstances, batteries replenishment can be difficult to attain and imply excessive costs. Hence, the need of an efficient network management strategy that saves energy to extend the network lifetime poses a primary need.

We consider a rectangular section of a river and we want to have early and reliable information about the presence of contaminants in the monitored area. Pollution stains can appear in any point of this rectangular section. The river dynamics consists of a flow that goes in one direction along the rectangle. Pollutants sediment and evaporate, which means that a certain portion of the pollutant is naturally removed. Another amount of pollutant travel downstream diffusing in the river and eventually occupying the whole section of the river. The model of diffusion we consider is a very simple one that takes into account the speed of the water flow and the notion of vicinity given by a grid model over the river. Pollutants in the river are subject to the river conditions. The contaminant stains may appear as a one shot event or with some periodicity. They may also last in time or be a continuous leakage. The intensity of the pollutant sources also conditions their presence in the river.

The goal is to obtain proper information from the environment that allows for a quick identification of pollution presence in the river. Although the scenario described may appear as a simple set, it includes the typical complexity associated to dynamic and continues systems. Moreover, it is general enough to allow for the extrapolation of the results derived from experimentation on this setting to similar problems but in different scenarios.

1.3 Proposal

The scenario considered for the research problem addressed is a river whose state has to be monitored. Different pollutant sources appear along the course of the waterway and their presence have to be identified. The deployment of a WSN in this kind of scenario permits the collection of information about the water state. This WSN can rely on the buoys and signalling elements, but also on other floatable devices specifically conceived for this purpose. The number of nodes to use and the cost of each sensor also conditions the network deployment scheme. However, the sensor nodes composing the network has to be able to sample the target environmental property and they need to be situated within each other sensing radius distance. That is, the network connectivity has to be guaranteed.

The river domain previously introduced represents a dynamic environment

due to the continuous water flow of this natural phenomenon and the contaminant stains. The state of the river can rapidly change, nonetheless periods of stable situations may happen. Besides, these stable situations or changes may be common to neighbouring nodes in the network. Therefore, it can be beneficial for nodes with a similar view of the environment to gather in groups and then sample and transmit data together. Joint actuation favours energy saving by avoiding redundant sensing and unnecessary transmissions. This reduction contribute to extend the lifetime of the nodes in the network. Nonetheless, this group organisation makes sense as long as the conditions for its formation hold. Moreover, grouping conditions must guarantee that this configuration does not detract the performance of the network.

The problem addressed in this thesis is the efficient and effective sampling of a waterway environment through local co-ordination of the nodes. The approach selected to accomplish this task relies on the achievement of a network organisational structure emerging from the network nodes co-ordination. To tackle this problem, we propose the *Coalition Oriented Sensing Algorithm*, COSA. The goal pursued by this algorithm is to extend the network lifetime while maintaining an adequate network performance.

This algorithm modifies the standard behaviour of nodes in a WSN by considering this as a *Multiagent System*. Each node is contemplated then as an intelligent agent capable of using the information perceived and received to negotiate with its neighbouring agents. Therefore, agents' activities are no longer limited to just sampling the environment and transmitting the data collected to the sink. Nodes use this information to communicate with their neighbour agents and establish relationships.

COSA defines a coalition formation algorithm based on peer-to-peer dialogues between neighbouring agents (nodes). Agents communicate to exchange information about their perception of the environment and their own state. As a result of this local communication, agents select the role to play in the organisation and can then establish *leader-follower* relationships. A *leader-follower* association is set when the two agents involved in a dialogue agree that one of them (the *leader*) will work on behalf of the other (the *follower*). On the one hand, the *follower* agent stops its sampling activity for a period of time. This halt allows the *follower* to save energy by not sampling, neither negotiating, during this period. Once this time has elapsed, a *follower* agent samples the environment again and uses this information to communicate its current perception to its neighbours in order to find its preferred role at that moment. A *leader* agent samples the environment regularly and informs the sink about the data collected. These tasks can be performed by and for itself (in case there are no *follower* agents associated to it), or on the contrary, it can act on the behalf of these *follower* neighbouring agents. Thus, through agents pairs co-ordination, the network configures itself in a set of groups that sample the environment as an entity.

The establishment of these peer-to-peer relationships relies on two relational functions whose values guide the agent behaviour (*adherence* and *leadership*).

The information exchange between agents is also governed by an operational protocol that lays down the norms of this co-ordination.

The *adherence* and *leadership* attitude concepts allow an agent to express its desire to form part of a group, and its capacity to lead a group correspondingly. The interest of an agent in forming part of a group led by a neighbour agent takes into account two factors. One of these factors considers the similarity of the agents' environment perception, that is, an agent wants to become a *follower* of a *leader* who samples similar values. Moreover, it wants the neighbour agent to have a 'good' model of the environment in terms of how informative it is.

The *leadership* attitude measures the capacity of an agent to act as a *leader*. Certain capacities or agent's attitudes are beneficial for playing this role. These factors are included in the *leadership* function definition to encourage the appearance of *leader* agents that verify them. Hence, the evaluation of the leadership attitude of a node takes into account if it is already playing this role; the energy available to perform this task; and the coherence of the group formed by those agents who are *followers* of it.

These functions definition requires the specification of a set of parameters that constrain the set of valid values that they can take. As a consequence, the agent's actions are also controlled. Thus, the application's performance requirements regarding the maximum size of the groups or the samples deviation for group establishment are introduced in the network function through agents' behaviour.

Finally, the agents' information exchange for a coalition definition is ruled by a simple negotiation protocol that governs agents' dialogues and which is entirely integrated within the agent regular behaviour. This protocol stipulates the existence of a set of performatives and the order in which they can be exchanged. Its reactive architecture based on independent rules allows the agent to take part in different dialogues at a time without causing deadlock situations. Moreover, the operation conditions guarantee the agent's actuation according to its preferences at any time and its adaptation to changes in the environment or network conditions.

The *Coalition Oriented Sensing Algorithm* proposed is completely embedded into the agent behaviour. Its adoption endows the network with a self-organisation capacity that emerges from the agents' local interaction and co-ordination.

1.4 Structure of the dissertation

Once the problem of interest has been briefly introduced, and the selected approach to provide an answer to it has also been outlined, the work developed to accomplish the corresponding research tasks is presented. This thesis is composed of eight chapters that cover from the review of previous works developed in the area to the evaluation and interpretation of the results derived from experimentation.

Chapter 2 focuses on the study of previous contributions on the problem

of interest. The review of the State of the Art is organised according to the two different approaches proposed for WSNs division into groups: the proper WSN community approach and the Multiagent Systems work.

The need of alleviating the constraints that WSNs present have favoured the interest of researchers from different research areas in order to promote the capacities of these networks. The Multiagent Systems (MAS) paradigm has contributed through the introduction of typical co-ordination and negotiation techniques used in agents' environments. The approach selected for the algorithm proposed in this thesis belongs to the MAS research area.

Previous works accomplished from the MAS point of view identify nodes of the network with agents in the system. This identification allows considering the nodes as autonomous agents capable of local interaction to originate a global behaviour at the system's level. The works reviewed take advantage of this perspective to propose different coalition formation strategies that allow the agents to perform tasks non-achievable individually or to do it in a more efficient way saving resources. As these preceding works, we propose a coalition formation strategy that, in this case, allows the agents to self-organise into groups to sample the environment and send the collected information to the sink.

The division of a sensor network into groups is referred as clustering in the WSN community. Work done in this line focuses mostly on routing strategies. As transmission is usually the most costly action, improving the route selection for transmitting to the sink generates important energy savings. Therefore, the definition of clusters that fix the transmission routes and that also adds the possibility of operation performance on the collected data have received considerable attention. Many algorithms with different purposes and using various approaches for cluster head selection have been proposed in the literature. A general view of these algorithms emphasising the differences with respect to the coalition formation strategy proposed in this thesis is presented in Chapter 2. Thereafter, the two research areas contributing previous works to the problem of interested are covered.

Detailed definition of the algorithm proposed to address the problem is presented in **Chapter 3**. *Coalition Oriented Sensing Algorithm* definition relies on two functions modelling the relationship between agents and a peer-to-peer negotiation process. The values registered by these relational functions condition the development of the negotiation process between a pair of agents, which may end up in the establishment of a *leader-follower* relationship.

The implementation of the rules and negotiation conditions in the agent defines its behaviour, that is, its operational protocol. COSA allows the agent to take part in different negotiations with distinct neighbours simultaneously without causing any synchronisation or interference problem. Agents just take advantage of local information to determine the role to play at a certain moment, consequently rising a global organisation structure based on local interactions.

The evaluation of *Coalition Oriented Sensing Algorithm*, as well as the assessment of its functional features, is based on simulation. The election of an appropriate simulator which offers the desired features is a relevant task. **Chapter 4**

motivates the choice of RepastSNS as the simulation engine for experimentation and presents this platform structure. RepastSNS, which is java based, and developed over a well-known Multiagent Systems' simulation platform poses as the best candidate. The study of its characteristics points out its main features, identifying their advantages and inconveniences when compared to the requisites for COSA experimentation.

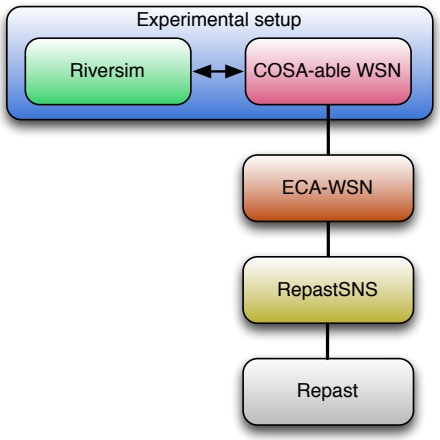


Figure 1.2: Development Structure.

Highlighted features of RepastsSNS are its general character and the communication capacity of every element modelled in the system. RepastSNS presents a low-level implementation of every element taking part in a sensor network. This feature represents a beneficial characteristic for the platform, as it provides it with a general and flexible character that makes it suitable for very different kinds of scenarios and problems. However, its generality, together with its novelty, demand an extra effort and an important amount of work to adapt the platform to the particular problem studied and to make it operate properly. Part of this work, as explained in the chapter, has been performed on RepastSNS and incorporated to its structure as it corresponds to general platform functions.

Figure 1.2 shows the development structure followed for the simulation execution. The structure is defined taking into account the scalability and extensibility features of RepastSNS. Thereafter, it presents a modular structure composed of different layers. Each of these layers corresponds to different implementation tasks that enrich the simulation platform towards the *Coalition Oriented Sensing Algorithm* application completion. Next chapters are dedicated to each of the layers built upon RepastSNS.

Chapter 5 presents the *Energy and Communication Aware Wireless Sensor Network* (ECA-WSN) layer which, as its name states, focus on primary aspects of the sensor nodes, such as energy management and communication activities.

RepastSNS does not assume any particular kind of environment, nor of the

sensor node. As a consequence, prior to the implementation of the MAS algorithm, the model of the network designed has to be built.

The energy management of a node refers to the consumption activities of the node's components and the battery operation. The energy policy implemented in this module meets RepastSNS's principles. Nonetheless, this policy provides with a completely different approach that standardises all consumption demands and considers the battery as an element capable of self-management.

Regarding the nodes' communication activities, the novel model issued by this layer allows for the emission of unicast and broadcast messages. Communication is a fundamental pillar of Multiagent Systems algorithms. Thus, an adequate model of this capacity is critical. The model of these new capacities and its corresponding activities is consistent with the energy model, what makes this layer a compact module modelling any regular sensor node's activities.

Chapter 6 describes the next layer of the structure designed for COSA simulation. ECA-WSN have considered physical aspects of the network nodes. The COSA-able WSN module abandons this device perspective to focus on the logical/behavioural dimension of the network nodes. That is, this layer properly implements the Multiagent Systems algorithm COSA. The implementation of the core of COSA behaviour into the agents adjusts to the formal definition presented in Chapter 3. The different aspects of COSA that contribute to the final concretion of the agent's behaviour are implemented in separate but linked structures.

An intelligent agent behaving according to COSA makes a conscious use of its resources. Regarding their power supply, agents can save a certain amount of energy to perform specific communication activities before completely depleting their batteries. Therefore, this COSA-able WSN module builds the corresponding energy and communication structures over ECA-WSN module to provide the network nodes with COSA specific functionalities.

This module design preserves scalability and extensibility features of the RepastSNS platform. Moreover, it also favours the implementation of potential algorithm enhancements and the reusability of the layered structure. In fact, this advantageous property is exploited to implement two alternative strategies to the original COSA formulation. Chapter 6 introduces the definition of two optional COSA strategies and presents the subtle implementation changes that they imply within the COSA-able WSN module.

Once the design of the network components has been specified, and their expected behaviour has been also codified, the last task to accomplish for COSA experimentation setup is the definition of the environment to be surveyed. This work belongs to the upper layer in Figure 1.2.

Chapter 7 presents the last layer built for COSA simulation. This chapter describes the experimental results derived too. The Riversim (*River simulator*) module contains the structure which specifies the application domain, particularly a river in which pollution sources appear and whose presence has to be warned. The model of this kind of environment bases on two components that represent the just mentioned elements, river and stain. The river compo-

nent mimics this natural phenomenon behaviour and stain components represent pollution sources poured into the river.

This environmental module is linked to the previously defined network through the specification of the environmental property that the sensor nodes can perceive. Sensors' particularisation to sample the environment defined by Riversim completes the network definition. However, simulation executions still required building up the simulation model which gathers together all the elements, and which is also in charge of correspondingly initiating them.

The experimentation accomplished aims at testing the algorithm performance. The experimental design done for this purpose takes into account features of COSA and the environment in order to define initial tests as complete as possible. The experiments' configuration proposed considers the same kind of node for every simulation but changes the network topology for different scenarios. Regarding the environmental conditions, the dynamics of the river and the stains also change from one scenario to another, as they represent different kinds of problems. The expected results from simulation are formulated in terms of hypotheses that are checked at the end of the chapter.

The evaluation of the potential benefits of COSA use bases on comparing the network performance when it implements COSA and a so-called Random policy. Random policy represents the typical WSN agents naïve behaviour of periodically sampling the environment and transmitting the collected sample to the sink. The confrontation of the results obtained from these approaches clearly manifests the impact of applying a MAS coalition formation technique in the WSN considered. The assessment criteria used for comparison contemplates the two main dimensions that characterise the performance of a monitoring WSN: the network lifetime and the correctness of the information reported.

Finally, the results delivered support the initial hypotheses about the expected behaviour of COSA and the alternative strategies proposed in Chapter 6. The comparison of the three strategies shows how the trade-off between energy consumption and information accuracy changes depending on the environment's conditions and makes one strategy preferable to the others.

Chapter 8 concludes this thesis by summarising the work accomplished. Conclusions about the algorithm performance and its implementation are finally drawn in this chapter. The benefits derived from more experimentation to fully characterise COSA are also recognised to encourage its use. Nonetheless, its implementation simplicity poses as an outstanding positive feature. The discussion about future directions of this research gathers enhancements and variation of the algorithm formulation, together with additional experimentation. These advances would lead to a methodic parametrisation of COSA. The availability of this methodology would allow an easy COSA adoption to potential application domains and, consequently, this would favour its use and spread.

1.5 Contributions

This thesis completion have resulted in the following contributions derived from the work developed.

The *Coalition Oriented Sensing Algorithm.* This algorithm appears as a self-organisation mechanism for WSNs, which aims at extending the networks' lifespan while meeting their primary goal, i.e. adequately monitoring the state of the environment. This strategy allows the network to regulate its energy consumption according to changes in the environmental conditions. The achievement of this behaviour stems from the recognition of each node in the network as an agent in a MAS. Hence, the design of an efficacious strategy for the network functioning focuses on the nodes' individual behaviour.

From this point of view, COSA sets the guidelines of a node behaviour in order to get a network-wide benefit. The definition of the node behaviour is specified by two functions and a negotiation protocol that allows the agent to establish relationships with its neighbours. Coalitions are formed as a result of these negotiations. One agent per coalition samples the environment and sends the perceived information to the sink on the group behalf, whereas the other group members stops their sampling actions.

The whole process relies on the local information available for each node at each moment. The use of this information according to COSA definition and parametrisation represents the way in which the programmer's interests guide the agent and the network behaviour. The network organisation in groups of nodes that act together to save energy takes into account each node state and its environmental perception. Thus, the information detriment caused by nodes' association is limited, and an adequate performance of the environment surveillance task can be guaranteed.

A workable version of RepastSNS. The verification of the algorithm functional properties and its study has been performed by simulating a particular instance of COSA in the target scenario described in Section 1.2. The simulation environment selected for this task is RepastSNS. In order to be able to use RepastSNS, the performance of a prior work on the simulation platform was necessary. The preliminary tasks accomplished have improved the platform operation and added new functionalities. As a consequence, we have contributed to the definition of a simulation environment specially conceived for WSNs study from a MAS perspective.

The setup of the platform and the implementation of our application over it follow a layered structure that constitutes a general development framework for applications over RepastSNS. Thus, the different elements taking part in the simulation, as well as different aspects of it, are introduced independent and orderly. This approach favours an easy substitution of the corresponding component modules to model different environments or circumstances.

A reusable generic model of a node physical behaviour. The first level built on RepastSNS is ECA-WSN. This module concentrates on physical characteristics of the node. ECA-WSN establishes an energy management model that allows for a homogeneous view of every energy consumer element in the node. Moreover, it also enhances the communication module with respect to the RepastSNS's original one. The notion of this module allows for its reuse in different applications, as well as its substitution in case the physical model of the nodes implementing COSA changes.

A modular implementation of COSA. The definition of the COSA-able WSN module transfers COSA formal expression to the simulation environment. This module uses the structures defined by ECA-WSN to code the agent behaviour following COSA. The modular implementation of the algorithm eases the introduction of changes and its replication in different frameworks.

Insights on the relationship between local co-ordination and energy saving. The results of the experiments performed bear witness of the utility of local co-ordination when the observed phenomenon also presents this character. It is important to remark this conclusion as the adequacy of COSA use particularly depends on the local/global nature of the phenomenon. Nonetheless, proper characterisation of the phenomenon is crucial for COSA adoption and the analysis of its behaviour.

Chapter 2

State of the Art

Wireless Sensor Networks (WSNs) are generally formed by a large number of sensing nodes able to develop surveillance tasks in different environments [Akyildiz et al., 2002]. Their typical features make them appropriate for an outstanding amount of monitoring applications [Martinez et al., 2004]. As a result, WSNs have been applied to different domains, such as environment monitoring, security and traffic control, target tracking, etcetera. Depending on the application environment and its accessibility, the challenges posed by these systems can be more or less acute, especially those referred to energy expenditure. Multiagent System (MAS) technologies can help in alleviating nodes' constraints referred to communication, processing capacity and energy availability [Lesser et al., 2003]. The introduction of coordination mechanisms between sensors can make the system, as a whole, more efficient in terms of energy consumption.

In this chapter, we review previous contributions on the area that have inspired our work. The point of view selected to study the aforementioned research problem differentiates previous works. The WSN community approach focuses on the development of clustering algorithms whereas the Multiagent systems' perspective concentrates on coalition formation among nodes in the network. In the following, we distinguish between these two research lines and review previous contributions to the problem considered.

2.1 Situation of the research problem

The problem tackled in this work is the development of an energy-saving data treatment strategy that, based on the local activity of the network nodes, permits an adequate performance of the system. Research works within the WSNs research area cover different aspects of it, but they all aim at improving the network capacity and attenuating the problems and constraints posed by the nodes' physical limitations and the deployment environments.

The work of [Yick et al., 2008] represents a very complete and concise survey

on the distinctive features of a WSN and its application domains. It also delivers a set of WSN features and processes whose improvement poses a challenge to developers. Authors divide the tasks to accomplish for a WSN creation into three main groups. From this distinction, and the five-layered communication protocol stack considered for WSNs, a survey on previous works unfolds. This survey describes the challenges to be faced at each stage of a WSN development. In any case, the importance of general aspects of the network, such as the need of prolonging the network lifetime and providing the agents with organising mechanisms is recognised throughout the whole paper.

The three group of tasks identified for a WSN creation are system's tasks, communication tasks and services tasks. System's tasks refer to the physical device; communication tasks cover activities performed within the base-mid layers of the communication protocol stack, and finally, services tasks comprehend functions related to the upper part of the communication stack and the network application. Among the open issues for a WSN creation pointed out within this classification frame, we can mention the development of more general WSNs' platforms that fit in different kinds of application domains, and the development of management and control services to deal with network connectivity issues. A deeper study of security, quality of service and mobility aspects of routing is also desirable, as well as the performance of cross-layer optimisation tasks. Sharing information through all communication layers leads to a more efficient network performance and an increase in the network lifetime.

Regarding the two previously mentioned research approaches identified for our problem (MAS point of view and WSN approach), these can be situated within the WSN accomplishment structure provided by [Yick et al., 2008]. The clustering work of the WSN community focuses on routing strategies, which belong to the network layer of the communication protocol stack, hence to the group of communication tasks. The implementation of coalition formation strategies between nodes can be classified as a service task. One of the goals established for these services tasks is to maintain the network operations, which matches the interest of works on coalition formation. Some of the strategies proposed to reach this target are synchronisation or data compression. The works of [Lasassmeh and Conrad, 2010] and [Srisooksai et al., 2012] present recent surveys on these strategies' development and application for WSNs. Synchronisation and data compression strategies pursue energy conservation and minimisation of the errors committed. Energy management is a critical issue that affects the whole WSN conception, from its design phase until proper operation. Energy harvesting [Basagni et al., 2013] and energy conservation [Anastasi et al., 2009] techniques try to maximise the network lifetime by allowing the replenishment of nodes' batteries or by introducing efficient strategies for energy use.

A general classification of energy conservation techniques is introduced in [Anastasi et al., 2009]. This work explores the principal contributions in this area by dividing them into three main lines: duty-cycling techniques, data-driven approaches and mobility-based strategies. Our work is part of the second group of techniques as it tries to avoid unnecessary sampling actions. To pursue this ob-

jective, the algorithm that we propose makes nodes implement an asynchronous sleep/wake up strategy, which is related to the duty-cycling schemes too. Therefore, our proposal could be considered as situated somewhere in the middle between the duty-cycling schemes and the data-driven approaches. Data-driven approaches are further divided into two subgroups, which are data-prediction and energy efficient techniques.

Data-prediction bases on the creation of a model for the observed phenomenon. This model can be used by the sink to predict the value of the samples instead of demanding a sampling action from the nodes. Some inconveniences presented by this kind of techniques refer to their specific character, as they highly depend on the application and the phenomenon of interest. The work of [Rogers et al., 2008] proposes an iterative formulation of a multi-output Gaussian process that can build a reliable probabilistic model of the environmental parameters being observed by a sensor network. This process is executed by a PDA (personal digital assistant) that has access to the data collected by a weather sensor network. The execution of this algorithm endows the element with the capacity of data prediction and autonomous data acquisition from the nodes, what allows it to determine when to read a sample and from which sensor. Nonetheless, energy efficient acquisition techniques aims at restricting the number of sampling actions. These strategies postulate as one of the most promising areas in energy conservation. They reduce the energy consumption by lowering the number of samples collected and thus, avoiding the energy cost of their associated transmissions to the sink. Our work pursues these objectives by benefiting from spatial phenomenon correlation. A similar perspective of the environment shared by neighbouring nodes in a WSN favours the possibility of these nodes group reunion. Nodes joint together share resources and sample the environment as an entity, a cluster in WSN community terms. The delegation of sensing and transmission tasks on the leader or cluster head allows the rest of group members to save energy. The algorithm that we propose to achieve this goal is based on the consideration of a WSN as a MAS. Thereafter, each sensor node can be contemplated as an agent able to communicate and establish agreements with its neighbours. The work of [Vinyals et al., 2011] presents a survey on MAS applications to sensor networks. This work identifies the sensor networks' problems addressed by the MAS community. It also presents promising research problems related to MAS application to WSN. Particularly, it points out collective sensing strategies as possibly one of the most promising areas. Numerous works have recently taken advantage of the adequacy of modelling networks with MASs. As a sample of this vein, the work of [Rebollo et al., 2014] proposes the use of a MAS to manage the dynamic demand in a power net. In the following, we review contributions done to the coalition formation problem from a MAS point of view and clustering strategies in WSN.

2.2 Coalition Formation in MAS

Coalitions represent a fundamental form of organisation for MAS. Agents cooperate within the coalition in order to share resources or reach shared goals that cannot be achieved individually. Coalition Formation (CF) has traditionally been studied from a game theoretical perspective as stated in [Horling and Lesser, 2004]. This work presents a review of organisational mechanisms and characterises each of them, among these mechanisms, coalitions. Nonetheless group formation of agents that model a sensor network function has been called coalitions, teams, regions, etcetera depending on the application domain or the authors considered. We assume the definition of a coalition previously introduced, as a group of agents that pursue a common goal. According to [Vig and Adams, 2007], CF in MAS can be studied from three different perspectives:

- *Task allocation.* Many MAS applications require agents to join forces for a period to solve a task. Contract Net Protocol [Smith, 1980] represents one of the first proposals in this line. The work of [Shehory and Kraus, 1998] continues in this line proposing algorithms for task allocation among agents.

- *Social networks.* This research line uses coalitions to study the emergence and behaviour of organisations in environments without clearly defined interaction mechanisms [Gasser, 1993]. SODA methodology [Omicini, 2000] exploits the social perspective of MAS for the analysis and design of Internet-based systems. Nonetheless, the work on CF from a social perspective have evolved towards a game theoretic approach.

- *Game theory.* This approach to CF has not traditionally focused on the design of agents' strategies to reach a beneficial coalition but on the study of stability and fairness properties of the coalition. The work of [Elkind et al., 2013] represents a very clear and complete survey on the concepts, representation formalisms and solution algorithms for coalition formation problems. The works of [Dang and Jennings, 2006, Rahwan and Jennings, 2008] represent recent approaches to the problem of coalition structure generation from this point of view. A dynamic coalition formation mechanism for distributed agents that model a smart electricity grid is introduced in [Mihailescu et al., 2011]. The mechanism presented focuses on the notion of stability associated to coalitional games.

The application of MAS techniques to the problem of coalition formation for effective sampling in WSNs demands an adaptive strategy. WSNs are typically deployed in dynamic and distributed environments whose state changes continuously in time. Therefore, there is no time, neither computation capacity in the nodes for a centralised approach to evaluate the optimal solution. Hence, a number of alternative mechanisms for coalition formation in this kind of environments have been proposed in recent years. According

to [Zambonelli and Omicini, 2004], MAS technology provides an effective way to solve the complex problems arising in this particular kind of scenarios that presents geographically distributed tasks and that also requires an adequate workload distribution of the tasks to be performed.

As stated above, agents' association to perform a task has been considered almost from the initial conception of the MAS paradigm. The approach taken for the design of these coalition or group strategies have evolved as the MAS application domains diversified. Therefore, a whole range of different CF mechanisms exist depending on the conditions and characteristics of the application scenario and the nodes composing the network.

A typical mechanism for coalition formation in MAS is negotiation. In the work of [Kraus et al., 2003], self-interested agents negotiate and reach decisions for task completion in a business inspired scenario with incomplete information about the environment and the other agents. These negotiations develop in so-called Request For Proposals domains. In the considered scenarios, business agents tackle complex decomposable tasks that require the formation of groups of provider agents to solve them. In contrast to them, we consider a network formed by cooperative agents whose common task is to monitor the environment. The interest in accomplishing this task efficiently is what drives the network division into groups.

The influence of the network topology on the performance of a MAS for task solving has also been considered in different approaches [Gaston and desJardins, 2005, Barton and Allan, 2007, Glinton et al., 2008]. In these cases, the system divides itself into disjoint groups in order to accomplish the demanded tasks. In the work of [Gaston and desJardins, 2005], agents can rewire their connections to neighbours to form better coalitions. This can be done according to their degree of connectivity or a performance-based policy. The decision factor for rewiring in [Barton and Allan, 2007] is the similarity among neighbours and some task and group success indicators. Finally, the work of [Glinton et al., 2008] enriched the previous one by considering a more realistic coalition model. All these works show how dynamic CF can improve the network performance. However, none of these three approaches takes into account the energy consumption and the cost derived from the rewiring policies.

Task oriented CF in dynamic environments faces the problem of high power and bandwidth consumption due to continuous configuration and reconfiguration processes to adapt to the system's evolving conditions and demands. To avoid that excessive consumption, [Bai and Zhang, 2008] proposes a task oriented team formation in which the group duration is calculated following some fuzzy rules applied to the historical behaviour of the agents and the characteristics of the tasks arriving to the system. A social model of coalitions also favours the appearance of mid-term duration coalitions [Griffiths and Luck, 2003]. The *clan* concept proposed in this work designates a set of agents with similar aims and mutual trust. In this case, group formation is not only determined by task accomplishment, but by the agents' motivation and trust relationships. The CF

strategy that we propose is based on a similar environmental perception and the agents' state. These metrics capture the interest of an agent in forming part of a group. However, we are not interested in keeping a coalition configuration when its origination conditions are no longer valid. Hence, coalitions' duration in our scenario is given by these conditions' time persistence.

The work of [Sims et al., 2003] presents two general CF algorithms for the problem of vehicle tracking by a sensor network. In the same vein of us, the proposed algorithms enable the self-organisation of the system by allowing the agents to discover their organisational relationships during negotiation processes. The CF process is modelled as a market environment, and negotiations can be based on local or social marginal utility calculations. The goal of the agents is to maximise the system's global utility, which is a function of the number of agents per sector and how well they cover the target region.

The Dynamic Regions Theory [Ruairi and Keane, 2007a] proposes a completely different approach for the division of a WSN in charge of a gas plume detection. According to this theory, the network partitionates itself into several different regions. The network partition is derived from the individual nodes' role election according to their circumstances and the system's global policy. Each region performs different tasks in the network. In contrast to this approach, we propose an algorithm that divides the network into groups that share the same responsibility with respect to the global objective of the network.

A more recent work that also introduces a strategy for sensor network division is [Bicocchi et al., 2012]. The network division algorithm is presented as an organisation mechanism specially conceived for pervasive sensor networks that do not have a fixed sink, and that can be accessed for information demand at any point. The identification of regions in the network relies on the differences registered for the observed variable. Each region is considered as a macro sensor able to report the value registered in its region. There is not a formal group establishment as there is neither task delegation. The scenario of this work is completely different to our application domain and goals pursued are also different. However, the interest for sensor networks division techniques and their application to different domains becomes clear in this work.

2.3 Clustering strategies in WSN

As previously discussed in Chapter 1, controlling the energy consumption of a WSN is a key issue. Clustering represents a prime technique to reduce the energy consumption of these networks and to extend its lifetime consequently. Clustering techniques have been typically implemented for routing algorithms and also as a basis for WSNs self-organisation mechanisms, especially for unstructured networks [Siham et al., 2013]. Dividing a network into clusters favours scalability and efficient use and sharing of resources. These properties translate into network topology stabilisation and energy saving. Clusters are formed by a set of agents in which one of them plays a special role, the cluster head. A cluster head can perform different functions, such as information aggregation or organ-

ise cluster members actuation to avoid collisions, for instance. Clustering also helps in diminishing communication overhead, decreasing packet collision and reducing the size of routing tables; that is, a more efficient use of resources.

There exist numerous contributions to clustering algorithms for WSNs. The work of [Abbasi and Younis, 2007] represents a quite complete survey on these techniques. It includes a taxonomy of clustering attributes that is used later in the paper to characterise the different algorithms reviewed. This work considers scalability as the prominent advantage offered by clustering. Hence, it presents the set of revised clustering algorithms emphasising whether they converge in constant or variable time. Besides this review, the survey also includes two useful tables. The first one compares the algorithms in terms of different metrics such as energy efficiency, load balancing, cluster overlap, stability, capacity of failure recovery, location awareness, node mobility and convergence time. The second table included summarises the reviewed algorithms' characteristics according to the proposed taxonomy, what helps in fixing the concepts and understanding the algorithms function. The proposed taxonomy considers three different groups of attributes: the first group focuses on cluster properties; the second group describes the cluster head capabilities, and finally, the last set of properties considered refers to the clustering process. Attending to this characterisation criterion, the algorithm that we propose for dividing the network nodes into groups presents the features included in Table 2.1. Some of the characteristics mentioned in Table 2.1 point out the MAS approach of our proposal, as its distributed character and cluster head selection through negotiation processes that end up with a cluster establishment.

A more recent survey on the area is presented in [Mitra and Nandy, 2012], which pays special attention to clustering algorithms for heterogeneous WSNs. Other works dedicated to clustering algorithms review have focused on energy efficiency, such as [Kumar et al., 2011] and [Siham et al., 2013]. The work of [Kumar et al., 2011] briefly review an important number of algorithms, including a quite exhaustive identification of LEACH algorithm descendants. On the other hand, [Siham et al., 2013] includes some more recent algorithms in its review, and a classification of WSNs' attributes similar to the presented in [Abbasi and Younis, 2007].

Among the most prominent clustering algorithms proposed for saving energy in WSNs, we can mention LEACH [Heinzelman et al., 2000], EEHC [Bandyopadhyay and Coyle, 2003] and HEED [Younis and Fahmy, 2004]. All these algorithms divide the sensor network distributively into a set of non-overlapping clusters. Each of these groups presents a cluster head which is in charge of sending the collected data in the group to the sink. Our approach differs from these works in the way the cluster head is chosen. The characteristics of the node, its state and the perception that neighbouring nodes have of it are taken into account when characterising a cluster head.

A more recent approach to this problem is presented in [Cordina and Debono, 2009], where a cluster-based routing algorithm is introduced. In this case, the base station determines which are the cluster

Table 2.1: COSA characterisation in terms of the taxonomy proposed in [Abbasi and Younis, 2007]

Category	Characteristics	Value
Cluster properties	Cluster count	variable
	Intra-cluster topology	fixed
	Inter-cluster connectivity	direct link
	Stability	adaptive
Cluster head capabilities	Mobility	fixed
	Node types	homogeneous
	Role	relaying
Clustering process	Methodology	distributed
	Objective of node grouping	energy saving
	Cluster head selection	negotiation
	Algorithm complexity	n/a

heads, and it also implements a centralised predictive filtering algorithm for decrementing the amount of transmitted data. In contrast, we propose an approach in which the nodes make autonomous decisions without any centralised control. Maintenance of the cluster structure is not significant for our algorithm as we encourage adaptation in its design. Nonetheless, recent approaches to clustering, such as the APC-T algorithm [Siham and El Ganami, 2012], promote cluster stabilisation to extend the network lifetime through load balancing. Moreover, this algorithm addresses one of the open issues pointed out by [Yick et al., 2008], the mobility of nodes. The cluster stabilisation is guaranteed by a cluster head replacement policy that the cluster head itself applies when its energy drops to a minimum threshold or it is going to leave.

In the vein of reducing the number of transmissions, but far from the coalition/group perspective presented above, the work of [Padhy et al., 2006] proposes an algorithm for individual node adaptive sampling that tries to extend the network lifetime of a glacial sensor network. This same goal is also pursued in the work of [Dyo et al., 2010]; that focuses on the importance of optimising the design and use of a sensor device for lowering the system's energy cost in the deployment of an automated wildlife monitoring system.

In contrast to these previous works, we propose a coalition formation strategy for homogeneous nodes in a sensor network scenario that allows to extend the useful lifetime of the network by avoiding redundant sensing and transmission. This group formation strategy bases on the nodes' state and the conditions of the environment. There is no intervention of any central authority and the algorithm

is fully distributed and embedded into the nodes' behaviour.

The main objective of reducing the energy costs of properly monitoring the target environment is achieved through the delegation of agents' sampling tasks to other group members. The selection of group members that share a common view of the environment becomes crucial in reducing the information loss in the process. Thus, the initial purpose of the system —faithfully monitoring the environment— is not missed.

2.4 Conclusions

In this chapter, works that recognise the importance of the research problem tackled and the need of solutions' proposal have been presented. The identification of a WSN with a MAS favours borrowing ideas from both areas for the definition of an algorithm able to respond to the problem requirements. Thereafter, different clustering strategies for energy saving in WSNs have been studied. CF algorithms in MAS have also been reviewed as they constitute an important mechanism for self-organisation and a tool for the system operation improvement. The study of works done in these two research areas provides the basis for the algorithm proposed in this dissertation. To the best of our knowledge, no previous work has addressed the problem of extending a WSN lifetime while guaranteeing its performance in a purely self-organised way. COSA allows for the network self-organisation through its division in groups. The network configuration is reached thanks to the common interest of cooperative agents in monitoring the environment effective and efficiently. The algorithm that agents implement to do this relies on a particular model of the agent's preferences for CF. The functions of this model take into account the similarity of the agents' environment perception, the environmental model of the nodes, the availability of energy, the perception that neighbouring nodes have of a node and the characteristics of the group to be formed. The value of these functions condition the execution of a peer-to-peer negotiation protocol that leads to the establishment of *leader-follower* relationships between agents. The duration of these relationships depends on the persistence of the conditions that favoured their appearance. Thus, the network configuration evolves and adapts to changes in the environment and in the nodes composing the network in order to avoid redundant sampling and provide the sink with adequate information.

Chapter 3

Coalition Oriented Sensing Algorithm

The standard and simplest behaviour of a sensor in a WSN consists of sampling the environment according to a pre-established frequency and then transmitting the data to a server, where this information can then be further processed and analysed. When the environment does not change, this behaviour wastes energy as many sensors will be transmitting the same data to the sink, and when the environment changes it does not adapt by increasing the frequency of sampling to provide better information. The objective of the *Coalition Oriented Sensing Algorithm* (COSA) proposed here is to improve this situation radically. To reach this goal, COSA subtly provides the WSN with a structure.

The basis of COSA arises from a social conception of WSNs. One of the key aspects of MAS studied from this point of view refers to the relationship between the social-organisational structure of the system and the autonomous agents composing it [Conte and Castelfranchi, 1995]. Considering a WSN as a social system, favours the identification of individual sensors with agents in a MAS. Each of these agents accomplishes particular functions in the system and in the society they define. From this social perspective and, assuming a simplistic approach to our problem, we can say that the individual behaviour of the agents and the interactions among them cause the emergence of new properties at the system's level. Hence, agents' low-level relationships lead to higher order links among them, what provides the network with an organisational structure.

The core of COSA lies in the establishment of groups among agents. To define these coalitions, and the role the agents play within them, agents negotiate through the exchange of information by short-distance communication. Thus, the resulting coalition structure depends at any time on the network topology, the state of the agents and the environment. As WSNs are deployed in dynamic environments, the distribution of roles among agents in the system will change along time. The use and interpretation of the information an agent has about itself, about its neighbouring agents and the environment are the key activities

of COSA.

3.1 Problem formalisation

The *Coalition Oriented Sensing Algorithm* (COSA) has been designed considering a scenario composed of a set $A = \{a_1, \ldots, a_N\}$ of cooperative and homogeneous agents (the network's sensors). We do not consider that agents can be competitive or selfish as in the kind of problems considered, there are neither resources to fight for nor rewards to be won by the agents. The pursued objective is that of improving the systems' global performance by enriching the capacities of the individual sensors.

WSNs are deployed in targeted environments in order to collect information from this area of interest. In such scenarios, the basic behaviour of an agent a_i is to sense the environment and relay the observed measures to a server or *sink*.

The goal of COSA is to save system's resources through coalition formation among agents that are perceiving similar measurements. Thereafter, a single agent can act as a representative of the coalition, avoiding redundant sensing and saving resources. To find an appropriate distribution of the agents in coalitions, we take into account the similarity of the individual measurements and the topology of the neighbourhood structure, which determines the neighbourhood relationships to be established among agents.

The unit distance assumed for this scenario is one radio hop. Let $d : A \times A \rightarrow \mathbb{N}$, be the distance between two nodes, measured as the minimum number of radio hops between them. The physical properties of wireless communication guarantees that d is a metric distance. In particular, d is commutative, $d(a_i, a_j) = d(a_j, a_i)$, and $d(a_i, a_i) = 0$.

Based on d, and given a set of agents A, we call $Ne : A \rightarrow 2^A$ a *neighbourhood function* if and only if $a_j \in Ne(a_i) \Leftrightarrow d(a_j, a_i) = 1$.

A coalition structure can then be defined for a maximum distance θ among the members of a group. That is, for $\theta = 1$, only direct neighbours can take part in a same coalition. Higher values of θ let agents whose neighbourhood relationship is at most, of θ order join in a group. For instance, if $\theta = 3$ then, a set of agents can form a coalition as long as none of them is further away from any other member of the group than 3 radio hops. Therefore, θ parameter influences the maximum achievable size of the coalitions, and hence the minimum granularity of the network. Given a set of agents A, a *θ-distance coalition structure*, $c_\theta = \{g_k\}_{k:1..K}$, is a partition of A in K coalitions, such that $\forall a_i, a_j \in g_k,\ d(a_i, a_j) \leq \theta$. We note by C_θ the set of all possible θ-distance coalition structures and the current coalition structure at time t, as c^t [1].

The criterion that guides the formation of the different coalition structures is to find (in a distributed manner) the best partition so that the energy consumption of the system is somehow minimised while the accuracy of the information sent to the sink hardly deteriorates.

[1] θ parameter is equal to 1 for the scope of this thesis

COSA appears as a tuneable algorithm thanks to the definition of a set of parameters p whose values drive the agents' behaviour. Depending on p, agents take different sampling and *transmission actions*. This set of actions is represented as $m^j \in M_p$, where M_p is the set of existing actions available for that p configuration. The right values' selection of these p parameters leads to the energy savings desired by the algorithm.

The objective of minimising the system's energy consumption is formally expressed in Equation 3.1, where m_i^j is the action j taken by agent i and E_j represents the energy consumption associated to that action.

$$p* = \arg\min_{p \in P} \Delta E = \arg\min_{p \in P} \sum_{m^j \in M_p} \sum_{a_i \in A} \#m_i^j E_j \tag{3.1}$$

This set of p parameters plays a dual role in COSA definition. According to Equation 3.1, we can find a set of parameters $p*$ that minimises the energy consumption in the system for a considered interval time. Besides, the accuracy of the measurements collected is also guaranteed through an adequate p parameters election as they constrain the actions to be taken by the agents. Nonetheless, to understand the behaviour of the proposed algorithm and the consequences of the individual agents' actions completely, the behaviour of the system is globally evaluated in terms of remaining energy, together with information entropy per unit time and joint error committed. Knowing that these are the global evaluation measurements, the behaviour of the individual agents and the effect of p parameters on it can be identified in the next sections.

3.2 Agent's coalition formation

COSA modifies the standard sensor behaviour (sampling and transmitting the collected information to the sink) by considering each sensor as an autonomous, proactive and reactive agent. To achieve this behaviour, COSA relies on a simple negotiation protocol and two functions modelling graded relationships: *adherence* and *leadership*. The numerical degrees of these relationships determine the asynchronous dialogues in which nodes engage when negotiating. At the same time, the results of these negotiations also modify the value of the adherence and leadership relationships.

As a consequence of this negotiation process, agents assume one of two possible roles: *leader* or *follower*. An agent is a *leader* if it is the representative of its coalition (where it may be the only member). A *follower* agent is that that joins a coalition led by another agent. The role performed by each agent in the system, and the link established with its neighbours is what defines the organisational structure of the WSN at a certain time. Therefore, through the individual agents' role election, the network configures itself as a set of coalitions that define a coalition structure for a specific instant of time. Each coalition in the coalition structure acts as an entity. The leader of the coalition senses the environment and sends the collected sample to the sink on behalf of the whole group. If this value is a good representative of the monitored variable in the area

covered by the coalition then, the coalition as a whole saves energy and computational resources. The addition of these coalitions' energy savings translates into a global system energy preservation. However, the possibility of reducing the energy expenses of the network actions come at the cost of detriment on the amount of samples available for the sink.

3.2.1 Relational functions: adherence and leadership

As already said in previous sections, coalition formation is the key element of COSA. Coalition formation is based on a peer-to-peer negotiation protocol by means of which agents exchange information about their measurements and their adequacy to represent their neighbours. These agent's attitudes are evaluated through the *adh* and *lead* functions, correspondingly representing the *adherence* and *leadership* concepts.

Given the set A of agents, the value of the functions $adh : A \times A \to \mathbb{R}$ and $lead : A \times 2^A \to \mathbb{R}$ change along time. Their evaluation depends on the value observed by neighbouring agents for the monitored variable at a certain time. In this work, we assume that the variable under observation follows a Normal distribution, $\mathcal{N}(\mu, \sigma)$, as this distribution represents a common model of natural phenomena [Manning and Schütze, 1999]. In this same vein, and in order to be able to evaluate its adherence and leadership attitude, each agent a_i assumes an initial model for the phenomenon observed, a Normal distribution $\mathcal{N}_i$. Individual agents update the value of their initial model as they collect new samples from the environment.

Adherence function

The *adherence degree* of an agent a_i to an agent a_j is a measure that indicates the intention of agent a_i to take part in a coalition led by agent a_j. The higher the degree, the higher the intention. The *adherence degree* is defined as the product of two factors. Each of these factors takes into account one of the two elements involved in the agent's information collection, which are the collected sample itself and the Normal model assumed by the agent. The first factor considers the similarity between the values observed by the agents, whereas the second factor evaluates the *certainty* that the neighbouring agent a_j has about its variable's model.

On one hand, the first factor in the adherence expression (Equation 3.2) captures the similarity between the measurements of agents a_i and a_j. It is defined as the quotient of the probability that the sample of an agent comes from the neighbour's distribution divided by the maximum probability reachable for that distribution. This factor's numerator expresses how likely it is that an agent's sample comes from a neighbour's distribution. The higher the probability, the similar the agents' perceptions are. This value is divided by the probability associated to the mean of the distribution in order to obtain an absolute measure. The first factor of the *adh* function is a ratio that does not depend on the specific variable model of an agent.

To avoid unproductive calculation, this factor is only defined for neighbour agents whose measurements verify that $\|x_i - x_j\| \leq d_{max}\sigma_j$; that is, the difference between the agents' samples (x_i, x_j) is less than or equal to the product of the parameter d_{max} by the neighbour's model deviation σ_j. This condition relates the difference between the samples taken by the agents to the shape of the distribution used for comparison as this absolute difference can be more or less significative depending on the kurtosis of the distribution.

On the other hand, the second factor captures the *goodness* of the neighbour's distribution, i.e. how informative the distribution is. To assess this property, we evaluate the entropy associated to the distribution, H_j, and normalise its value in the interval of interest. As the observed variable is assumed to follow a Normal distribution, the entropy evaluation is based on the standard deviation of the function ([Goldman, 2005]). In these conditions, we define an interval of interest for the distributions based on their standard deviations, $(\sigma_{min}, \sigma_{max})$. σ_{min} represents a very low value that just defines the lowest extreme of the interest interval. Values of σ higher than σ_{max} are associated to wide distributions that lead to coalitions with disperse information. To avoid this situation, a null *goodness* value is associated to neighbours whose deviation verifies $\sigma \geq \sigma_{max}$. This condition is introduced in the second factor of the *adh* function through the constants H_{min} and H_{max} associated to parameters σ_{min} and σ_{max}. The definition of this factor also uses exponential functions. The shape of the exponential function results especially interesting for normalising the entropy value of an agent's variable model, as its value quickly grows when approaching the selected upper extreme. This second factor can be interpreted as a modulator of the first one. Its value ranges from 0 to 1 so that the adherence takes the first factor value for agents whose σ is near σ_{min}, whereas this factor is strongly penalised for agents whose σ is situated around σ_{max}.

Finally, the evaluation of the degree to which an agent a_i may be interested in being led by one of its neighbours a_j is calculated as follows:

$$adh(a_i, a_j) = \frac{p(x_i, \mathcal{N}_j(\bar{x}_j, \sigma_j))}{p(\bar{x}_j, \mathcal{N}_j(\bar{x}_j, \sigma_j))} \cdot (1 - \frac{e^{H_j} - e^{H_{min}}}{e^{H_{max}} - e^{H_{min}}}) \tag{3.2}$$

These two just presented multiplying factors can be identified in Equation 3.2. To sum up the role each of these factors play in the *adherence* definition, it can be said that the nearer the sample to the neighbour's mean is the higher the adherence whereas the thinner agent a_j's model of the monitored variable is the more adherence.

Leadership function

When an agent a_i receives an adherence value from a neighbour a_j, it has to decide whether it is interested in becoming the leader of this agent or not. To determine its *leadership* degree, agent a_i does not only take into account the adherence message just received from this neighbour a_j. In this evaluation, a_i also considers the adherence values previously received from all other nodes

currently 'depending' on it (including itself). That is, those agents assuming the follower role and considering agent a_i as their leader.

A good leader has to comply with three requisites referring to different aspects of the task it may develop. First, it should be a good representative of its follower neighbours in terms of phenomenon characterisation. It is also necessary to have enough energy to sense and communicate with the sink. Besides, to become the leader of another agent, it is beneficial to be already performing this role for other follower agents.

Let's call potential group, $P(a_i)$, to the set of agents composed of the new neighbour a_j and the follower agents associated to a_i (including the own a_i). Then, the willingness of a_i to act as a leader of a set of agents $P(a_i)$, depends on three factors that can be identified in Equation 3.3. The names given to these factors refer to the corresponding aspect that they represent.

The first factor, which is called *prestige*, is an average of the adherence level of the members of $P(a_i)$ towards a_i. This factor captures how an agent is perceived by the other agents in the network. Taking into account how much 'wanted' as a leader an agent is favours promoting the leadership attitude of that agent.

The second factor, *capacity*, considers the available energy of the agent to act as a leader. This value is derived from the current energy level of the agent, $E(a_i)$, minus the so-called *Energy security level*, E_{sl}, over the maximum energy level of the battery, E_{max}. E_{sl} represents the amount of energy necessary to send one last disconnection message to warn the network about the node's exhaustion.

Finally, the last factor, *representativeness*, indicates how well agent a_i's measurement fits as a representative of the potential group agents' measurements. Thus, a_i characterises the set of samples of agents in $P(a_i)$ with their mean and standard deviation, noted as $(\bar{x}_{P(a_i)}, \sigma_{P(a_i)})$. To encourage the formation of coalitions with very similar measurements, an exponential function establishes the divergence growing ratio. Those potential groups whose measurement distribution is very disperse are also penalised through the inclusion of the Pearson's coefficient ($CV_{P(a_i)}$) in the equation. Equation 3.3 presents the leadership capacity of an agent a_i for a potential group $P(a_i)$:

$$lead(a_i, P(a_i)) = \frac{\sum_{a_j \in P(a_i)} adh(a_j, a_i)}{N} \cdot \frac{E(a_i) - E_{sl}}{E_{max}} \cdot \frac{1}{e^{|x_i - \bar{x}_{P(a_i)}| CV_{P(a_i)}}} \quad (3.3)$$

As previously explained, COSA is designed with a set of parameters that establish a limit to the computational effort of each agent in the evaluation of these relationships. Hence, this set of parameters, p, influences the actions that an agent can take. At this point, we have already introduced five parameters composing p. Therefore, the set p can now be identified as $p = \langle \theta, d_{max}, \sigma_{min}, \sigma_{max}, E_{sl} \rangle$ defined over the space $p \in \Re^5$.

The first parameter included in p, θ, constrains the size of the coalitions to be formed, as it determines the maximum distance, in radio hops, admissible between agents in a coalition. The second parameter, d_{max} puts a limit on the maximum difference between samples to evaluate the adherence to a neighbour j,

$\|x_j - x_i\| \leq d_{max}\sigma_j$. This maximum difference is proportional to the neighbour's σ_j to take into account the shape of its distribution.

The extreme values of the interval $(\sigma_{min}, \sigma_{max})$ establish the limit for the variance of the set of coalition members. The smaller the gap the more demanding we are in terms of the similarity of the coalition members' measures. Hence, more coalitions of smaller size would be formed. The larger gap, the dissimilar node values in a coalition can be. This would imply larger errors at the sink but larger and more stable coalitions (i.e. more energy savings). Finally, E_{sl} parameter conditions the energy available for the node to act as a leader, and it also allows it to send a last disconnection message.

3.2.2 Operational Protocol

The adherence and leadership degrees registered by agents drive their actions to form coalitions in the system. Agents' preferences change due to the dynamics of the environment and the dynamics of the individual sensors. Therefore, based on these instantaneous values, agents negotiate trying to achieve their most preferred configuration at each time.

The default and initial situation of the network corresponds to every agent in the network alone constituting a coalition by itself (led by itself). In this case, the agent, which is its own leader, senses the environment according to the demanded sampling period and sends this information to the sink.

Algorithm 1 represents the general behaviour of an agent implementing COSA. The *Sense And Send* thread represents the basic behaviour of a network node slightly modified to include basic characteristics of COSA. However, the real potential of COSA is introduced in this simple thread of action by the addition of the *Information Processing* thread.

Algorithm 1: Agent generic behaviour

```
Data: t(sampling period), t'(sleeping time)
spawn (InformationProcessing);
while E(me) > 0 do
    if (!leader) then
        stop (SenseAndSend(t));
        stop (InformationProcessing);
        sleep(t');
        resume (InformationProcessing);
        spawn (SenseAndSend(t,role));
    else
        spawn (SenseAndSend(t,role));
kill (InformationProcessing);
kill (SenseAndSend(t,role));
```

Implementing COSA into an agent behaviour demands the previous specifi-

cation of the parameters $p = \langle \theta, d_{max}, \sigma_{min}, \sigma_{max}, E_{sl} \rangle$. Parameters defined by the monitoring application also need to be specified, such as the demanded sampling period and the sleeping time allowed for the agents. The agent's behaviour is organised around two main threads of actions, which are *Information Processing* and *Sense And Send*, and their execution depending on the role played by the agent at each time.

When an agent starts working, it initiates the thread *Information Processing*. As long as it has energy, and depending on the role assumed by the agent, it executes the actions corresponding to the *Information Processing* and *Sense And Send* threads. As it can be seen in Algorithm 1, when an agent assumes the leader role (which is also its default role), it samples the environment regularly and sends this information to the sink.

Algorithm 2: Sense and Send

```
Data: t(sampling period), role(leader or follower)
next = now;
if next=now then
    sampling;
    modelUpdating;
    if role=leader then
        sinkTransmission;
    neighboursBroadcast;
next=next+t;
```

The thread *Sense And Send* collects the sample from the environment according to the demanded sampling period. This piece of information is used for two tasks as it can be seen in Algorithm 2. The first task consists of updating the variable model of the agent, and the second one refers to the transmission of the sample. To comply with COSA requisites, individual agents use their samples to update their models of the environment as explained in Section 3.2.1. As the variable follows a Normal distribution ($\mathcal{N}(\mu, \sigma)$), updating this model with this new piece of information implies updating the value of its corresponding mean and standard deviation. This way, the agent perception of the environment evolves in time.

The second task consists of transmitting the collected sample. Every agent collecting a sample from the environment, whether it is a leader or a follower, takes advantage of this piece of information to find its most desirable role at that time. To do this, it sends to its neighbours the just collected sample. Besides this, a leader agent always transmits the latest sample to the sink. In the case that there is a set of follower agents associated to this agent, that is, it is not a single agent coalition, sending the collected sample to the sink implies transmitting information on behalf of all those follower agents. The set of follower agents of a leader is called *dependent group*. If the leader agent is alone, it sends the sample just on its unique representation. A leader agent is

also continuously executing the *Information Processing* thread and consequently, negotiating and exchanging information with its neighbours.

The first set of actions appearing in Algorithm 1 corresponds to a follower agent role. A follower agent stops sampling the environment and negotiating with its neighbours during the sleeping time. Once this period is over, awaken agents recover their capacities. At that moment, it can sample the environment and use that information to negotiate with other agents (leader or recently awaken agents). That is, follower agents can take part in the coalition formation process right after awakening. This way a follower agent can change its relationship with its associated leader. It can become the leader of itself or other agents, it can become a follower agent associated to a different leader, or it can continue with its previous relationship. This decision is always determined by the current conditions at that time. The *Information Processing* thread is the element that supports the negotiation process and that guides agents' actions and role changing depending on the current conditions.

As it can be inferred from the preceding explanation, COSA protocol is embedded into the agent behaviour mainly via the execution of the *Information Processing* thread and the actions corresponding to the agent role at any moment (leader or follower). The role developed by an agent changes along time depending on the information available at each moment, that is, collected and processed through the *Information Processing* thread.

Negotiation protocol

Before going into the details of the *Information Processing* thread, we are going to present the basis of its functioning. The introduction of the main concepts that have driven its design help in a better understanding of the conditions and consequences it defines.

The working regime imposed by COSA to the agents can be divided into four processes that run simultaneously:

- *Sample information exchange.* This process corresponds to the variable sampling and measurements broadcast. Once the agent has taken a sample from the environment, it communicates this information to its neighbours in order to find suitable coalition members.

- *Adherence relationship establishment.* Once the agent has calculated the adherence degrees to its neighbours, it communicates the maximum adherence value to the corresponding most preferred neighbour.

- *Leadership information exchange.* Based on the current adherence relationships, the agent calculates and communicates its attitude as a leader towards those agents willing to adhere to it.

- *Coalition definition.* Depending on the information available for an agent at a certain moment (own state, adherence degree, own leadership degree and neighbour's leadership degree), it decides whether to stay in its current

coalition (as a leader or follower of a leader agent), to leave this coalition to join a different one, or to constitute its own coalition.

The messages the agents exchange during the negotiation follow a classical agent communication format: *performative(sender, addressee(s), [msgContent])*. A communication message is specified by its kind (performative), the agent initiating the message (sender), the agent to which the message is sent (addressee) and, finally, the content of the message (*msgContent*). This last field is optional and is only included in those messages with information associated. We consider the existence of five different performatives for negotiation (each one corresponding to a different kind of action). The set of performatives that agents use are:

- *inform*: performative used to indicate the transmission of data (samples, maximum adherence or leadership values). Values associated to these variables are sent in the *msgContent* field of the message.
- *firmAdherence*: performative used to express the desire of the sending agent to adhere to the addressee one and to establish a follower-leader relation with it.
- *ackAdherence*: performative used to express the acknowledgment to a previously received firmAdherence message.
- *break*: performative a leader agent uses to break a leadership relationship with a follower agent.
- *withdraw*: performative a follower agent uses to break a leadership relationship with its associated leader agent.

Note that, within the first three processes listed before, only the *inform* performative is used as the three of them consist of a different kind of information exchange. The other performatives presented are used within the coalition definition process, whether to establish new relationships or to dismantle existing ones. Some of these performatives can also carry additional information in their *msgContent* field. This information refers to the current leadership attitude of the agent. The performatives that include this additional data are *inform* and *ackAdherence*. The goal of adding this piece of information is to guarantee that the relation established, or to be established, bases on up-to-date information. Therefore, we take advantage of the emission of these messages to transmit information about the leadership attitude of the sender. Implementation details of the negotiation process are precisely presented in Chapter 6.

Finally, one performative extra, not directly related to the negotiation processes, is introduced. The *death* performative represents the message transmitted by an agent when it is about to die. This message emission is possible thanks to the E_{sl} energy level introduced in COSA definition, which establishes the need of energy preservation to accomplish this task before complete battery exhaustion. The usefulness of this performative is presented together with the description of the *Information Processing* thread in the following section.

Information Processing thread

The basic idea of the negotiation protocol is quite simple. When an agent samples the environment, it sends the observed value to its neighbouring agents. An agent receiving a sample from a neighbour uses this information to evaluate the adequacy of forming a coalition. If this evaluation is positive, the neighbour agent sends an adherence message back to the agent initiating the dialogue. The agent receiving this adherence message offers itself to work for the two of them (assuming the role of leader of the coalition). If an agreement is reached, the leadership offer is accepted, then the agent sending the sample becomes the leader while the neighbour, which assumes the follower role, sleeps and stops its sampling and sink transmission tasks for the sleeping period. Therefore, through peer-to-peer negotiations between neighbours, agents select their preferred role and build a coalition structure in a bottom-up fashion. The set of figures presented in 3.1 represents this process for a pair of agents. In the case depicted, the negotiation goes through all existing stages and finishes with an agreement between the two agents. The agent initiating the dialogue assumes the leader role while its neighbour becomes its follower.

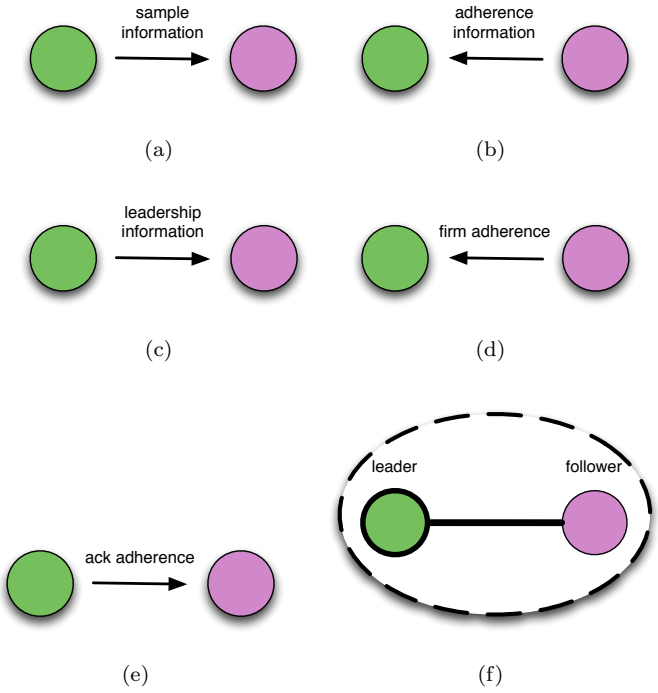


Figure 3.1: Negotiation protocol stages.

Adopting the best organisation translates into energy savings by avoiding unnecessary sampling and long-distance transmissions: those of the follower agents.

This apparently straightforward scheme quickly complicates as the number of nodes in the network grows. For a simple scenario of three agents, the system may end up in one of ten different configurations. Figure 3.2 shows an example of these possible coalition structures. Alternative configurations not depicted can be obtained just by swapping the leader/follower roles between agents in a coalition, what eventually renders a different network organisation scheme.

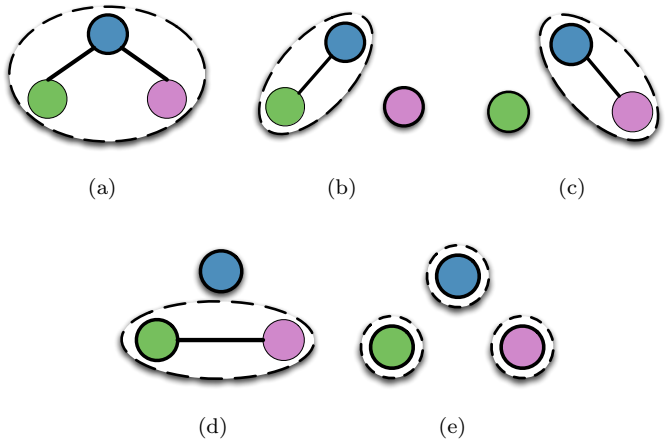


Figure 3.2: Possible coalition configurations.

COSA protocol is embedded into the agent generic behaviour. Agents behave in a proactive and reactive way. Proactive because the core behaviour of an agent is the continuous process of looking for the best group of neighbours that matches with its measurement and its state. In order to achieve this objective, an agent exchanges messages asynchronously with its neighbours. Agents behave also reactively because their acts and decisions are triggered by the observation of the environment and the information they receive.

The core of the coalition formation process is contained in Algorithm 3. This algorithm has been designed following a simple reactive structure in which all the actions are triggered by an event that changes the available information for an agent.

The first attempt to implement this algorithm in the agent behaviour was based on the Finite State Machine model. Describing the agent behaviour as a sequence of stages and conditional-transitions led the system to high complex interactions and undesired deadlocks. Therefore, we shifted our approach to a reactive architecture composed of a set of simple independent rules. This perspective allows the agent to decouple the different negotiation dialogues it may be involved in and also the different stages of each process. From a global point of view, this way of interpreting the information and acting consequently can somehow be comparable to the existence of an information flow among the agents, being this flow responsible for the agents' reaction.

When a message is received, depending on its kind, its processing may imply the update of the internal model of the agent about its leadership value and about the model associated to the agent sending the message. Depending on these updates a decision is made that may imply new messages being transmitted back to the sender or to other agents in the network. The code, as can be seen in Algorithm 3, allows for the intermingling of dialogues with different neighbours.

As previously said, agents implementing COSA exchange information via performatives and using a classical alternating negotiation protocol. All these performatives are used in Algorithm 3, and the meaning of each procedure is rather self-explanatory.

The first three processes correspond to the reception of the same kind of performative: *inform*. However, as each of this *inform* messages presents a different content in its *msgContent* field, its processing differs and origins different kind of actions.

The first procedure corresponds to the reception of an inform message containing the sample collected by a neighbour together with its perception of the environment. Therefore, this procedure represents the first step of the negotiation protocol: an agent sends its sample to its neighbours. Receiving this information causes the agent to update the information it has about its neighbour's model, and also, to take advantage of it by evaluating the adequacy of forming a coalition with this neighbour. The agent evaluates its adherence degree to this neighbour. If the resulting adherence degree is the maximum value reached at that moment, the agent sends back an *inform* message containing the value of the adherence degree and its environmental model. In case the adherence degree obtained of this evaluation does not represent an improvement with regards to the existing value, no message is sent.

The second procedure represents the set of actions to be taken when an *inform* message containing data about the adherence degree of a neighbour is received. These actions are aimed at the treatment of a message resulting from the execution of the previous procedure by a neighbour agent. In this case, the content of the *inform* message gives information about the desire of the neighbour agent of forming a coalition with the addressee. Receiving a message from a neighbour indicating its adherence degree implies sending an answer back to tell this neighbour about the agent's attitude as its leader. This can be interpreted as the emission of a leader offer to a potential follower. The rest of the procedure repeats the same actions discussed when an *inform* message containing a sample is received (first procedure). When an agent sends an *inform* message to inform about its adherence degree towards a neighbour, it also provides the neighbour with information about its characteristics, such as its sample (the one used to evaluate its adherence degree). This sample information is used to evaluate the opposite relationship by the recipient agent. That is, the addressee agent checks if it is interested in forming a coalition with this neighbour, being led by the neighbour. As before, if the adherence value obtained is the maximum at that moment, an *inform* message containing the value of the adherence degree together with its identification information is sent back. In this case, a new

Algorithm 3: Information Processing

Data: me: focus agent; a_j: generic neighbour; a_l: potential leader; a_r: potential follower of me; a_p: follower agent of me; a_L: leader agent of me; $D(me)$: set of follower agents of me

case *received(inform($a_j, me, meas$))*
- updateNeighbourInfo(a_j);
- adherence2NeighbourEvaluation(a_j);
- updateOwnMaxAdherence();
- **if** *changesOnOwnMaxAdherence* **then**
  - inform($me, a_l, maxAdh, t$);

case *received(inform($a_j, me, maxAdh$))*
- inform($me, a_r, lead$);
- updateNeighbourInfo(a_j);
- adherence2NeighbourEvaluation(a_j);
- updateOwnMaxAdherence();
- **if** *changesOnOwnMaxAdherence* **then**
  - inform($me, a_l, maxAdh, t$);

case *received(inform($a_l, me, lead$))*
- **if** *checkAgainstOwnLead* **then**
  - firmAdherence(me, a_l);

case *received(firmAdherence(a_r, me))*
- **if** *checkAgainstOwnLead* **then**
  - **if** !*leader* **then**
    - withdraw(me, a_L);
  - ackAdherence(me, a_r);
  - updateRoleState(leader);
  - $D(me) \leftarrow D(me) \cup a_r$;

case *received(ackAdherence(a_l, me))*
- **if** $!leader \wedge a_l != a_L$ **then**
  - withdraw(me, a_L);
- **if** $leader \wedge D(me) != \emptyset$ **then**
  - **while** $D(me) != \emptyset$ **do**
    - break(me, a_p);
- updateRoleState(follower);
- sleep(t);

case *received(break(a_L, me))*
- updateRoleState(leader);

case *received(withdraw(a_p, me))*
- $D(me) \leftarrow D(me) \backslash a_p$;
- updateRoleState(leader);

case *received(death(a_j, me))*
- **if** $a_j == a_p$ **then**
  - $D(me) \leftarrow D(me) \backslash a_j$;
  - updateRoleState(leader);
- **if** $a_j == a_L$ **then**
  - updateRoleState(leader);
- updateNeighbourInfo($\overline{a_j}$);

dialogue between the two agents already negotiating initiates as a consequence of the already existing one. Nevertheless, the design and implementation of the protocol guarantees the absence of conflicts.

As explained in Section 3.2.2, an inform message can carry different kind of data in its *msgContent* field. When an *inform* message containing the leadership attitude of a neighbour is received, its processing follows the scheme presented in the third procedure. To decide if the addressee agent is really interested in being led by this neighbour sending the message, it compares the just received lead value to its own lead attitude, which may correspond to the agent itself or to its leader, if it plays a follower role. After this comparison, if the agent prefers to assume the follower role associated to this neighbour, it sends a *firmAdherence* message. In case that the agent prefers to continue in its previous state (the comparison of leadership degrees fails), this dialogue between these two agents finishes at this point.

The fourth and fifth procedures correspond to the processes that follow a generic successful dialogue initiated with a sample information exchange and that finishes with an agreement between the two nodes, one assuming the role of leader and the other one assuming the role of follower. When an agent which has previously sent a lead offer receives a *firmAdherence* message, the first thing that it does is to test if it is still willing to act as a leader for this neighbour. During the exchange of messages between these two agents, the agent's conditions can have changed and messages from other neighbours may have arrived. The addressee agent can be involved in more than one dialogue, each of it at its own stage, what causes changes in the agent state during this particular negotiation dialogue. However, if the agent decides to become the leader of the neighbour, it has to send the last message of the dialogue, an *ackAdherence* message. Before doing this, if the agent is a follower of a neighbour, it has to break this relationship with a *withdraw* message. As a consequence of this action, the agent updates its state as a leader taking into account its leadership attitude and the set of agents that, from that moment on are followers of it and constitute its dependent group.

Receiving an *ackAdherence* message implies the immediate assumption of the follower role by the addressee agent. That is, after processing this message, the agent deactivates its sampling and transmission actions and stops the execution of this *Information Processing* thread until the sleeping period has finished. Before reaching this state, and depending on the agent current situation and the relationships it already held with its neighbours, different actions may need to be performed.

The first condition of the procedure refers to the situation of an addressee agent which is not leader. That is, this agent is a follower of a neighbour, but the neighbour to which it is currently associated is not the same that sent the lead offer and this *ackAdherence* message being processed now. The situation described corresponds to that of an agent which wants to continue being a follower but now it wants to depend on a different leader. In these circumstances, the addressee agent needs to break the relationship with its current leader be-

fore going to sleep. To do this, it sends a *withdraw* message to its leader. After performing the corresponding required actions, the agent updates its state as a follower and initiates the sleep time.

A different situation that can also take place is that of the addressee being a leader agent with its own coalition, and also wanting to change its role to become a follower. A leader agent has to break the relationship with all its follower agents before it can become a follower itself. To do this, the addressee sends *break* messages to all those agents in its dependent group.

The sixth procedure represents the actions corresponding to the reception of a *break* message. This message can only be received by follower agents and its sender is the corresponding leader of the addressee agent. The action to be performed when this message is received is just the update of the agent state to leader. Hence, the recipient agent becomes active again. It then starts sampling the environment and negotiating with its neighbours in order to find the best configuration in its new circumstances.

Withdraw messages are sent by follower agents to their leaders. When an agent receives a *withdraw* message, this comes from an agent that was a follower of the agent and that now wants to break the relationship. The reception of this message implies the deletion of the sender agent of the set of followers of the addressee.

The last procedure appearing in Algorithm 3 shows the actions corresponding to the reception of a *death* message. As explained before in the previous section, this is not a proper negotiation message but its reception unavoidably affects the state of the agent's relationships. If the agent that is about to die is a follower of the addressee, it has to be deleted from the dependent group to register that this relationship does not exist anymore. As a consequence of this change, the addressee agent updates its state as a leader taking into account the existing relationships. In case that the agent that sent the *death* message is the leader of the addressee, the recipient changes its state from follower to leader. Regardless of the kind of the relationship held by the agents, the addressee deletes the sender's information from its register.

The detailed explanation of Algorithm 3 points out the actions to be performed by an agent involved in a negotiation process. This description allows for a better understanding of COSA functioning and how it affects the agent's behaviour. Moreover, this section completes COSA presentation by relating the concepts that define the algorithm to the agent's actuation.

3.3 Conclusions

COSA represents a MAS approach to the problem of lifetime extension of WSNs. This algorithm addresses redundant sensing situations that waste energy in sampling and unnecessary transmitting actions. In order to avoid them, COSA introduces an organisation mechanism within the agents' behaviour. This mechanism allows the agents to divide the network into coalitions and sample the environment together without causing significant deterioration of the network

performance. The fulfilment of these premises relies on a negotiation protocol that enables peer-to-peer co-ordination of the nodes. The protocol conception guarantees the absence of collisions when different dialogues overlap. An agent's preferences guiding the negotiation are determined by the evaluation of the *adherence* and *leadership* functions. The assessment of these parametric functions is based on the agent's local information. The concepts supporting COSA, as well as the behavioural strategies designed for the nodes, have been carefully described in this chapter in order to ease subsequent development and implementation tasks.

Chapter 4

RepastSNS simulator

As mentioned in previous chapters, the technological progress reached in different areas has allowed the development of smaller sensors at lower prices. This fact has significantly favoured the deployment of real sensor networks in different scenarios. Among typical application domains for these networks, we can mention security and safety, habitat and environment monitoring, biomedical applications, smart spaces and distributed robotics.

MASs have been successfully applied to sensor networks due to their capacity for naturally modelling a set of autonomous sensors physically distributed and their interactions [Sycara, 1998]. In most cases, an agent models a sensor node. However, the MAS configuration depends on the characteristics of the network itself and the approach selected by the designer for the tackled problem. Hence, MAS can be open, cooperative or competitive depending on the possibility of nodes' replacement or network purpose or ownership.

The identification between WSNs and MASs have contributed to the development of the WSN research area by introducing new methodologies in it, such as the computational theory of organisations, market mechanisms design, cooperation and coordination algorithms as well as distributed learning and algorithms for information distribution [Chen-Khong and Renaud, 2005, Ruairi and Keane, 2007b, Rogers et al., 2006].

However, despite the ease of development of real sensor networks and the growing availability of commercial platforms at affordable prices, simulation platforms are still widely used by researchers. The main reasons for this are the high costs derived from the deployment of a real network, the inherent variability associated to the results obtained in real scenarios tests and the impossibility of neglecting the network's low level details.

Research tasks include protocol design to improve the efficiency of communications among nodes, individual nodes' operation or the whole network management.

From the reasons mentioned above the main one for testing sensor network algorithms by simulation is cost. The deployment of a sensor network requires an important investment of money, whereas the use of simulation platforms to

study them is much cheaper.

The development of new protocols and behaviour strategies for the nodes in a network requires an iterative process of testing and refinement. Appropriate experimentation demands a controlled environment that ensures the repeatability character of the tests. This property cannot be guaranteed when the experiments run on a real sensor network as there are unavoidable uncontrollable factors.

Finally, the development of software research tasks directly on real sensors forces the designers to take into account all low-level and hardware details of the nodes. This represents a significant burden of work. Besides this, it is also an important drawback for the researchers as it prevents them from testing their algorithms at the desired abstraction level and the development time increases.

In these conditions, we plan to test our COSA algorithm on a simulation platform. Thereafter, in this chapter we briefly review some of the existing simulators and their characteristics. Then, we present RepastSNS, the simulation engine over which we will develop our application.

4.1 Sensor Network simulators

Nowadays, computer simulators are simple and efficient tools able to deal with the growing complexity of the systems being modelled. A basic classification divides them into continuous simulators and discrete simulators.

Continuous simulators require precise equational knowledge of the behaviour of the elements involved in the simulation in order to model them mathematically. The simulator applies the equations of the model depending on the conditions defined by the environment. However, the equations used to model a system in a continuous way are complex and the computational cost of their evaluation increases exponentially with the size of the system. This makes these simulators appropriate only for small scenarios with a low number of elements. Therefore, typical application domains of sensor networks requires the use of discrete computer simulation.

Unlike continuous simulators, discrete simulators do not use a mathematical formulation to describe the behaviour of the elements in the system. In this case, the state of the system changes at precise moments as a consequence of the triggering of events. Describing the system's behaviour consists then on identifying the sources of change (events) and defining the corresponding consequences to be executed. The event scheduling follows a simple scheme that consists of orderly adding the events to a queue and executing them in this same order at their corresponding simulation time.

4.1.1 Platform requirements

The characteristics of our target scenario and our protocol demand the use of an event-based simulator. However, the election of a simulator requires taking also into account other aspects that require from the simulator interesting characteristics:

- Open source: It is necessary to select an open source platform so that our work can be reproduced and used by the scientific community.
- General purpose environment: the abstraction level provided by the simulator has to allow for the instantiation of different kinds of sensor networks adequate for different domains and/or applications. To incorporate this feature, the simulator needs to provide an infrastructure that interrelates the basic components of a sensor network, such as observable phenomena, sensors, agents and communication mechanisms.
- Extensibility: the platform needs to facilitate reusing and extending its components for different simulations. This feature favours the study and characterisation of algorithms in different domains and for different purposes as it prevents high programming and configuration costs.
- Scalability: the key potential of sensor networks arises from the cooperation activities among the high number of nodes composing them. The platform has to manage the functioning of these nodes assigning corresponding resources to each of them.
- Repeatability: experiments executed using a simulation platform must always deliver the same results from a statistical perspective for the same initial conditions. This is a basic requisite to have a valid scientific software supported by the community.
- Time control: the application domains of sensor networks are typically dynamic. Thus, the algorithms designed for these systems do not only need to focus on finding the best configuration for the sensor nodes or the best strategy to fuse the information, but they also need to focus on the system's response time. As a consequence, accurately modelling the actions' duration and their interactions with the environment is one of the simulator's key requisites.
- Observation: the execution of a simulation produces information that has to be collected and correctly presented to the user. This information can be of two kinds: general and domain specific. General information refers to common parameters to every sensor network (energy consumption, nodes' position, etcetera). Domain specific information includes useful information to understand and evaluate the system's performance in a particular scenario (measures collected by sensors, content of messages sent or received, etcetera). The simulator has to integrate a set of tools that perform an automatic collection of general and specific information, together with a view or report generator.

4.1.2 Brief survey on sensor network simulators

The first simulation environments that appeared for sensor networks were an adaptation of more general simulators, such as *ns-2* [DARPA, 2013] or *J-Sim* [Sobeih et al., 2005]. New simulation environments especially designed for

sensor networks appeared later. Here, we review the characteristics of the most common network simulators.

ns-2

ns-2 is one of the most popular simulators in the field of computer networks. This platform is actively maintained and used, having evolved in the last years to support typical protocols of wired and wireless networks.

The simulator has been developed using C++ and OTcl. The platform's engine is based on discrete events and follows an object-oriented architecture, what favours extensibility [Sundani et al., 2011]. Nonetheless, ns-2 cannot model real time OS or code execution delays [Korkalainen et al., 2009]. Other inconveniences of this simulator refer to scalability and the time required to learn how to use it properly. Furthermore, this simulator does not have an application layer, what hinders the implementation of typical MAS data fusion or learning algorithms.

OMNeT++

OMNeT++ [Varga and Hornig, 2008] is a discrete event and component-based sensor network simulator. It is defined as a set of different modules connected in a nested hierarchy. Basic modules of *OMNeT++* are implemented in C++, whereas more complex structures use a proprietary language (NED) as a high level language.

OMNeT++ mainly supports IP networks, but there are extensions for WSN. This simulator is not as widely used as ns-2 [Varga and Hornig, 2008], although it scales well, presents an application layer and allows for a relatively straightforward components' modification.

J-Sim

J-Sim is a general purpose discrete event simulator initially conceived for computer networks [Sobeih et al., 2005]. However, it can also be applied to any system whose components' state changes at discrete instants of time. It is written in two languages (Java and Jacl) and presents a component-based design which facilitates extensibility and scalability features.

J-Sim provides support to different WSN elements (sensors, physical phenomena, etcetera. Nonetheless, it is complicated to use and only allows for the use of 802.11 MAC protocol [Sundani et al., 2011]. As it was not originally conceived for WSN simulation, its design makes it difficult to add new protocols or components, which represents an important inconvenience.

Radsim

Radsim (Radar Simulator) [Lawton, 2003] is a discrete event simulation environment that was conceived to test networks of cooperative agents. The simulator

was developed in Java and meets many of the requirements desirable for a simulation platform. However, it presents two important drawbacks: it is not a public platform, neither a general purpose sensor network simulator as it was conceived for tracking moving objects.

SENSE-Sensor Network Simulator and Emulator

SENSE [Chen et al., 2005] is a sensor network simulator that appeared in 2004 with the objective of becoming a powerful simulator easy to use.

The platform is implemented in C++ and presents a component-based design. This structure confers to the simulator a set of advantages referring to components reusability, good extensibility features due to loose coupling among components and parallel execution, which also reverts in a scalability improvement [Sundani et al., 2011]. However, this platform demands a low level approach, which is quite distant to the MAS point of view in which we are interested.

Repast 3

Repast 3 is a step-based simulation environment for MAS [Sourceforge, 2012]. It is an open source platform quite known within the MAS community. This simulation platform, which is Java implemented, provides a large set of adaptive algorithms. It also presents different utilities to register the simulations and analyse their results. There are also general packages available that provide additional functionalities for the simulations.

Repast 3 is a multithreaded environment, so that agents' actions are performed in a separate thread, and events are delivered continuously. The actions' duration can be specified before they take place. However, the multithreaded environment represents an important penalty with respect to execution time and, moreover it also hinders experimental repeatability.

The analysis of these simulators shows that none of them meets all the requirements desired to test our COSA algorithm. *ns-2* lacks an application model, *OMNET++*, *J-Sim* and *SENSE* do have this layer but they present a very low level approach that does not favour a MAS perspective. *Radsim* is not a public platform, neither a general purpose simulator and *Repast 3* does not offer enough control to model communications, nodes activity and energy consumption.

Given this, we decided to use the RepastSNS platform to test COSA algorithm. This platform results from the evolution of a previous work developed within the IEA project (Institucions Electròniques Autònomes). The study of sensor networks from a MAS point of view was the main aim of this project developed at the IIIA-CSIC. One of the outcomes of the project was the SNS platform [Pujol-Gonzalez, 2008]. The SNS is an event-based platform designed

to study sensor networks from a MAS perspective. This simulator provides a set of components that allows modelling sensor networks easily.

RepastSNS is the result of the effort of migrating the SNS platform to convert it into a Repast package. This work [Matamoros, 2008] was motivated by the interest of attracting users to the simulator. As previously said, Repast is a well known and established MAS simulation tool. Therefore, transforming the SNS sensor network simulator into a Repast package significantly favours its use and dissemination into the MAS research community.

As part of this intention, we run our COSA algorithm over RepastSNS. This circumstance poses us in a beta-tester position, as the function of this platform was only tested at a very low level with simple demonstration examples. Hence, this situation represents a double challenge for us, as we are going to develop and implement a real research application over a new simulator. This implies testing its usability and the satisfaction degree of its design characteristics reached. This process has allowed us to fix numerous bugs referring to different aspects of the simulator that lead to an homogenous structure of its components, a correct functioning of the scheduler and the simplification of the initial conception of some processes.

4.2 RepastSNS

RepastSNS [IIIA-CSIC, 2012] is an extension of Repast classes, so the program structure fits into this known MAS simulation engine. RepastSNS is an event-based simulator especially designed to model sensor networks as multi-agent systems. A distinctive characteristic of RepastSNS is that all the objects in the environment are modelled as agents able to communicate via message passing.

This open-source platform was developed in Java over Repast. Its structure consists of two layers: an object layer and a network layer. The object layer defines the behaviour of the individual nodes and the network layer defines the topology and the relationships among the nodes composing the network.

The lower layer provides the basic structure for a sensor network simulation, including all the needed objects to make an event-based simulation and also abstract classes for the objects composing a general sensor network. These classes, which are called base components, allow the user to control all generated events and their associated processing, to model hierarchical and control relationships among components and also to regulate communication. Figure 4.1 shows the elements composing this layer.

The upper or network layer contains instances of the base components of the object layer. This layer follows a component-based architecture in which each component shows its functionalities. Hence, the platform design distinguishes two layers each of them implemented according to a different approach, an object oriented architecture and a component oriented architecture. The structure of the network layer can be observed in Figure 4.2.

RepastSNS meets all the requirements desired for a simulation platform. It is an open source platform as it is a Repast package completely implemented

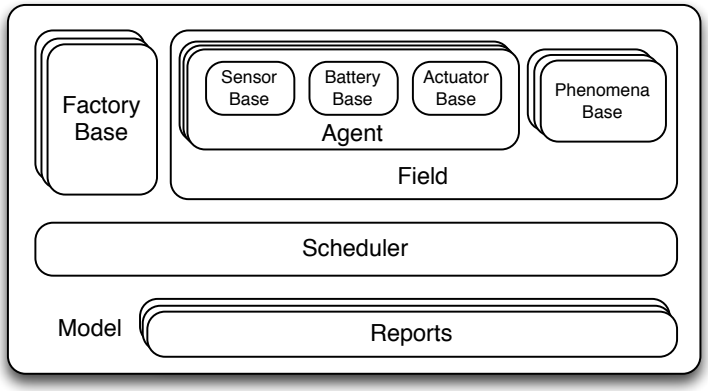


Figure 4.1: Object Layer [Pujol-Gonzalez, 2008].

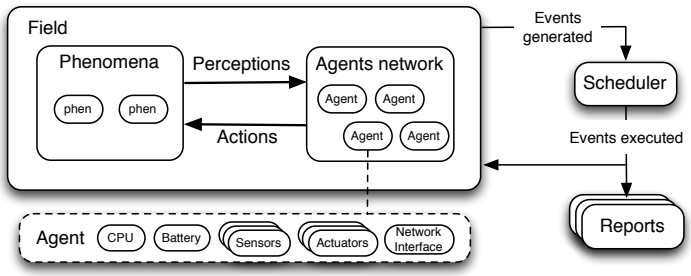


Figure 4.2: Network Layer [Pujol-Gonzalez, 2008].

in Java. Different kinds of WSN problems can be implemented and tested over this platform. The characteristic two-layer structure of the platform makes it a general purpose simulator, able to perform experiments on different scenarios. Furthermore, this structure of object layer and network layer makes the simulator extensible and scalable. The primary behaviour of each component is specified at the lower layer, whereas the relationship that components may establish among them are also clearly regulated at the upper layer. These relationships are based on the exchange of particular *events*, whose reception and transmission can be planned in time. Therefore, the platform offers time control on the actions performed in the system. All these events happening in the simulations can be detected and monitored, what renders in a high observability of the simulations performed. The repeatability of experiments is guaranteed by the element that hosts all other elements in the simulation, the *model*. The *model*, considered as part of the network layer, provides cohesion to all other elements and initiates them appropriately.

The advantage of using RepastSNS instead of any previously presented sim-

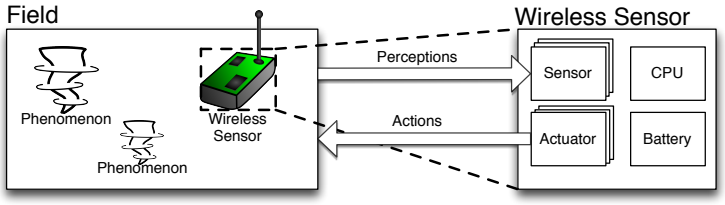


Figure 4.3: RepastSNS simulation architecture.

ulators is twofold. Firstly, it provides for a more abstract level description than these other simulators. Therefore, it allows the programmer to concentrate on the actual agent behaviour instead of dealing with hardware details. Secondly, it brings with it a convenient basic implementation of all the components needed to model wireless sensor networks. Thereafter, the central task for a user of this platform is to configure and adapt the platform's predefined elements to the particular domain in which he is interested.

Figure 4.3 outlines the architecture of a sensor network simulation on RepastSNS. In RepastSNS, all the observable phenomena are contained inside a *field* that includes the nodes themselves. Furthermore, the nodes are composed of multiple modules: a *CPU*, a *battery*, and any number of *sensors* and *actuators*. Sensors are those devices that allow the node to perceive the field's phenomena and their properties. Analogously, actuators allow the CPU agent to modify existing phenomena or produce new ones. This very simple model is surprisingly sound as any phenomena or agent behaviour needed in a system can be easily modelled and incorporated. For instance, wireless radio interfaces can be modelled as an actuator that generates wireless waves (a phenomenon), plus a sensor that detects them.

4.3 RepastSNS main features

RepastSNS defines a set of classes and interfaces for correct integration into Repast. As aforementioned, a particular concept of *event* allows for the communication among elements in the system. The transmission and reception of these events constitute a source of information about the simulation performance and consequently favour detailed observation of the system.

Another feature demanded to the simulation platform referred to the control of the simulation time. Hence, knowing the time at which these *events* are generated can give crucial information to the platform's users to understand the behaviour of the system. RepastSNS allows a temporal management of the events and actions taking place in a simulation, dealing with processes' duration or actions' delay when resources are not available. Repast does not present any of these capabilities; therefore, their incorporation required the addition of new functionalities to this simulation engine.

4.3.1 Simulation elements' communication capability

Communication among elements in the simulation scenario takes place through the exchange of a particular kind of *events*. These new *events* need to be generated by the simulation elements, scheduled by the platform and received and interpreted by the addressees elements.

Simulation Events

Those particular *events*, *SimulationEvents* can be of different types and have different functions. *SimulationEvents* allow for the communication among simulation elements, but they also report information about changes happening in the system during a simulation performance. *SimulationEvents* providing information about the simulation execution are grouped into three classes: *AddedEvent*, *RemovedEvent* and *PropertyChangedEvent*. The first and the second classes give information about the time and the object originating a simulation element addition or removal event, whereas the third one warns about the change of value of a property observed in the system. *SimulationEvents* used in communication vary depending on the elements exchanging information. Its definition is also conditioned by the nature of the information exchanged.

SimulationEvents' management for communication requires the simulation elements to implement the interface *SimulationListener*. This interface makes the simulation elements able to receive *events*. It also obliges the elements to implement the method *simulationChanged()*, which takes a *SimulationEvent* as parameter. This method also allows for the reception and processing of an event from a different element. This method processes the event received by searching for an adequate *process* method specific to the event received by the object. If the object does not have it (because it has not been specified by the designer), a default implementation of the *simulationChanged()* method is executed.

This process requires the scheduler of the platform to be able to pass a *SimulationEvent* to the object playing the receiver role in the information exchange. The scheduler provided by Repast, as in any other event-simulator, is in charge of setting the pace of the execution and controlling the simulation time. It executes the events orderly according to their time and controls the simulation time updating it to the value of the corresponding executed events. However, a distinctive characteristic of Repast is that it calls these events (scheduler basic units) *actions*. Therefore, making a change in the simulation requires building one of these *actions*.

RepastSNS presents the *BasicActionSNS* class. This class is an extension of the Repast class *BasicAction*, which defines the scheduler basic units. *BasicActionSNS* adds a *SimulationEvent* property, such that when this event is added to the action, the method of the destination object is executed taking this event as its argument (which represents a functionality that was not originally provided by Repast).

BasicActionSNS class extends from *BasicAction* and adds the properties *target*, *methodName* and *event*, which are, correspondingly, the destination object,

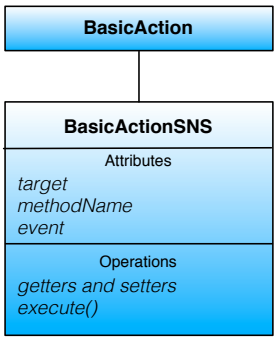


Figure 4.4: *BasicActionSNS* class outline [Matamoros, 2008].

the method of that object that we want to execute and the event handed as parameter. These three properties have their own getters and setters. The *execute()* method, given by the superclass *BasicAction*, continues being an abstract method to be implemented by the programmer. Figure 4.4 shows the structure of the resulting class BasicActionSNS.

Repast uses the classes *ActionUtilities* and *ByteCodeBuilder* to generate, dynamically, classes extending from *BasicAction* at runtime. This way, the *BasicActions* created can implement the abstract method *execute()* to call destination object's methods. The action creation structure presented in the RepastSNS package differs from Repast's one to allow the creation and execution of *BasicActionSNS*. These differences can be seen in the classes *ActionUtiliesSNS* and *ByteCodeBuilderSNS*. Changes basically refer to the overload of methods for actions' creation. Hence, *ActionUtiliesSNS* and *ByteCodeBuilderSNS* admit the creation of actions from *SimulationEvents*.

ScheduleSNS definition

The definition and creation of *BasicActionSNS* enables the communication among elements in the platform. However these actions need to be scheduled at their corresponding time for this functionality to work properly. RepastSNS manages this through its ScheduleSNS class.

The ScheduleSNS class introduces changes with respect to the Repast scheduler that affect the scheduling process at different levels. The first difference between both schedulers refers to the unit time used. RepastSNS uses nanoseconds as its basic unit time to improve the temporal management of the simulations.

A new functionality provided by RepastSNS which is not present in the Repast simulation engine is the capability of delaying actions. This functionality appears as a consequence of the communication ability of the elements in order to guarantee correct delivery. This functionality relies on the *lastWorkingTime* property of those simulation elements receiving a *SimulationEvent*. As it name states, the value of this property indicates the time instant at which the object finished or will finish its latest task.

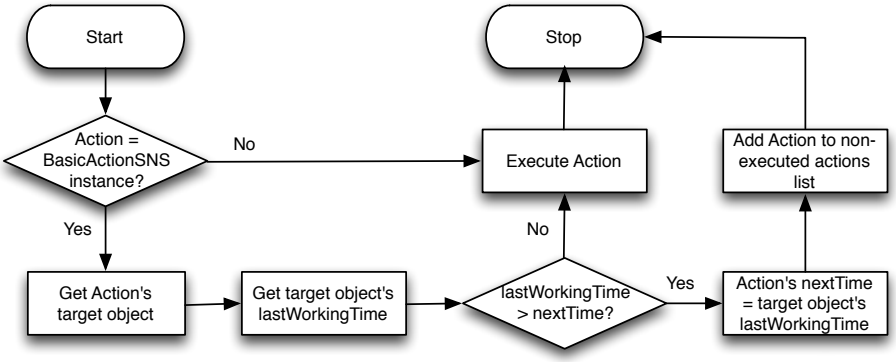


Figure 4.5: Algorithm for actions execution [Matamoros, 2008].

Communication among elements of the simulation allows triggering the execution of a method of the addressee object through the reception of a *SimulationEvent*. If the addressee object is engaged in a task, it cannot execute the demanded method when the *SimulationEvent* is received. Hence, the scheduler of RepastSNS delays the execution of this action until the object is available that is, until its *lastWorkingTime* value.

The core of ScheduleSNS follows the same structure as the Repast's scheduler. The scheduling relies on the *execute()* method of the ScheduleSNS class. This *execute()* method relies in turn on other methods and classes to perform its task. The classes it relies on are *ActionQueue* and *ScheduleGroupSNS*.

The *ActionQueue* class keeps the actions to be executed until their execution times arrive. The *ScheduleGroupSNS* class groups actions that have the same execution time. Hence, the *ScheduleSNS* executes groups of actions, not individual actions. In general terms, the working scheme of the *ScheduleSNS*'s *execute()* method is as follows: first, a *ScheduleGroupSNS* element gathers the actions to be executed at the next timestamp. Then, each of the actions in the *ScheduleGroupSNS* that can be executed is executed and, the ones that need to be rescheduled (because they take place periodically or there are not free resources for them to be executed now) are inserted again in an *ActionQueue* element. The capability of delaying actions is introduced in the last step of this process affecting the *ScheduleGroupSNS* class.

The *execute()* method of this *ScheduleGroupSNS* class, which is responsible for executing each of the actions it contains, follows the algorithm shown in Figure 4.5.

The algorithm begins by testing if the action is of *BasicActionSNS* class. If it is not, i.e. it is an original Repast's *BasicAction* then, it is executed as this new module would not exist. Hence, *ScheduleSNS* can schedule original Repast's actions without any loss of generality. On the other hand, if the action is an instance of *BasicActionSNS* class, then the target object is obtained to ask it about its *lastWorkingTime*. This property tells if the object is currently engaged

in a task or not. If the addressee object is available, the action is executed. If it is busy, the action's execution is delayed until the target object is available, that is, the action's execution time is set to the value of *lastWorkingTime*. Then, as already explained, the action is added to the corresponding *ActionQueue* element actions not executed yet.

Besides this, *ScheduleSNS* class presents the methods *addMessage* and *addEvent* which complete the whole integration of the communication events in the system as they allow for the scheduling of actions created from communication events. Figure 4.6 shows the basic structure of *ScheduleSNS* class.

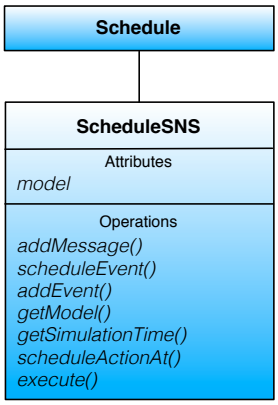


Figure 4.6: *ScheduleSNS* class outline [Matamoros, 2008].

4.3.2 Model: simulation environment cohesion and experiments repeatability

As it was already mentioned when the platform architecture was presented in Section 4.2, the model component is the element which gives cohesion to the whole system. The corresponding object in RepastSNS is the *SimModelImplSNS* which brings together the functionalities provided by Repast and the ones required by the RepastSNS platform itself.

Following the definition scheme of Repast classes, *SimModelImplSNS* implements the interfaces corresponding to the functions it will provide. Hence, it implements the *SimModel* interface (Repast standard), the *SimulationListener* interface (communication capability) and *SimModelSNS* (functionalities of the RepastSNS platform).

Thanks and through the model, all elements which are model listeners get to know about the state of the simulation. This is how objects receive information about what is happening during the simulation. As the model itself implements the *SimulationListener* interface, it can also receive information from other simulation elements.

The *SimModelSNS* knows all the elements in the simulation. It is in charge

of setting up and initialising all of them. Every simulation element implements these two methods which are executed by the model through its method *prepareElement()*. The *SimModelSNS* is also in charge of generating *long* parameters handed to the corresponding elements' *setup()* methods. This random number is used as the seed for the elements that need it. This guarantees the repeatability of the experiments as two runs of the same simulation will give the same results.

As an element which assembles together all the elements in the simulation, it keeps a reference to the main components presented in Figure 4.2, such as *ScheduleSNS*, *SimulationField* and *SimulationComponentFactory*. The *SimulationComponentFactory* allows the generation of new simulation elements, such as agents or phenomena, during the simulation. These references and their corresponding getters and setter methods can be observed in Figure 4.7.

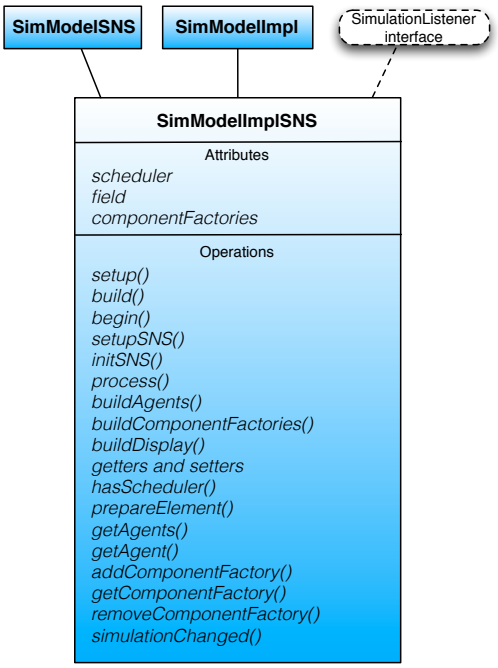


Figure 4.7: *SimModelImplSNS* class outline [Matamoros, 2008].

4.3.3 Simulation observability

Another additional feature presented by RepastSNS is the *SimulationEvents' table*. This table (shown in Figure 4.8) allows the user to see a set of the last executed *SimulationEvents* ordered by time. Each *SimulationEvent* is represented by a horizontal register in the table, which shows the time at which the *SimulationEvent* occurred, the element which originated it and finally, a description of the *SimulationEvent*. This tool is especially useful for debugging tasks.

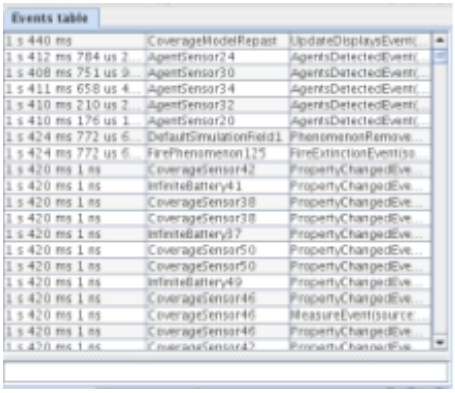


Figure 4.8: *SimulationEvents* Table.

Besides this, it also considerably increases the observability of the system, as a detailed report of all the *SimulationEvents* happening in the system is available for inspection.

The *SimulationEvents' table* is provided by the classes *SimulationEventTableView* and *SimulationEventTableRenderer*, which are specific of RepastSNS. This table implements the *SimulationListener* interface. Therefore, it presents the method *simulationChanged()*. As explained in Section 4.3.1, this method allows the object to receive and process *SimulationEvents*, process that in this case consists just in attaching the *SimulationEvents* to the *SimulationEvents' table*.

The introduction of this table in the Graphical Interface of the platform and the capture of these *SimulationEvents* for monitoring purposes differentiates the Graphical Interface of RepastSNS from Repast's. The *ScheduleSNS* class is also affected as it has to deliver the *Simulation events* for two different purposes: the *SimulationEvent*'s original one and the attachment to the table.

4.3.4 Simulation elements identification

The introduction of the *SimulationEvents' table* allows for a better understanding of the platform working scheme. Nonetheless, to follow the trace of an agent is a difficult task.

The component structure definition of the platform, which confers scalability and extensibility features upon it, prevents tracing the behaviour of a particular agent. Elements composing an agent establish relationships between them through messages exchange. These elements may or may not know the existence of the other ones. However, the programmer cannot easily know wether two elements are related and make up the same agent, as each kind of element (sensors, actuators, etcetera.) is named independently. As a consequence, when an action message is emitted by a transmitter, for instance, we cannot identify the node (agent) emitting that message, but only the transmitter element.

To resolve this situation we add an identification variable, a label called *id* at

the lowest level of the elements structure, that is, to the *AbstractSimulationElement* class. When an agent is created, the value of this identifier is set. This same value is assigned to all the other agent's components, such as the transmitter or the receiver. Hence, all the elements composing a node share the same value. The use of a unique identifier common to all the elements composing a node eases the previously mentioned activities and makes the platform more accessible and easy to use.

4.4 Sensor Network simulation elements

Another advantageous feature of RepastSNS is the existence of a basic implementation for typical elements appearing in a sensor network, such as sensors, actuators or phenomena to be observed.

As RepastSNS is java-based, basic definition of these elements is given by an interface and the corresponding abstract class associated to each element. This architecture eases the addition of new elements to the simulation and, of course, it contributes to the flexible and adaptable character of the platform.

4.4.1 The field

WSNs are typically used to monitor the state of a phenomenon of interest in a particular environment. Therefore, there must be a representation of this environment in the platform.

The *SimulationField* interface represents the physical environment where phenomena to be observed appear and in which the WSN is deployed. That is, this interface represents the space where all the physical elements of the simulation interact. Figure 4.9 shows the structure of this interface.

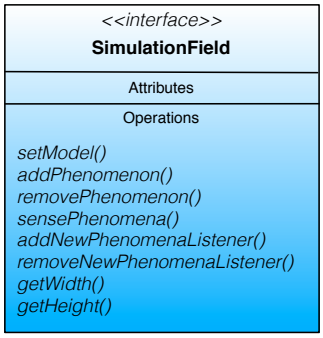


Figure 4.9: *SimulationField* interface outline [Pujol-Gonzalez, 2008].

This interface includes the methods *addPhenomenon()* and *removePhenomenon()* that allow for the addition or removal of a phenomenon in the *field* respectively. As a consequence, the *field* knows which phenomena are active at each time. Its corresponding abstract class is responsible for notifying

the system about this fact by sending the corresponding *Simulation Events*, in particular *PhenomenonAddedEvent* and *PhenomenonRemovedEvent*. As their names state, the *PhenomenonAddedEvent* informs about the addition of a new phenomenon to the *field*, whereas the *PhenomenonRemovedEvent* notifies the disappearance event.

Simulations in RepastSNS consider that all existing phenomena are situated in the *field*, that is, they have a presence in the *field*. Hence, agents are also considered as an observable phenomenon. However, agents represent a special kind of phenomenon as they are also composed of other elements. As explained in previous sections, agents include sensors and actuators through which they can get information from the environment and also act on it.

Therefore, the *field* performs a special role in the simulation. It is the only element in the system which is aware of all existing active phenomena and it is in charge of linking the agents and the phenomena. The way this task is performed depends on the kind of sensors modelled by the agent. However, the *SimulationField* interface includes different methods to accomplish this task: *sensePhenomena()*, that returns a list of all the active phenomena that a sensor can perceive; *addNewPhenomenaListener()* and *removeNewPhenomenaListener()*, which allows a sensor to subscribe (unsubscribe) to a particular kind of phenomenon in the field. As a consequence of the establishment of this relationship, a sensor receives information about the phenomenon's appearance and disappearance.

4.4.2 Phenomena

RepastSNS considers as a phenomenon in the system every occurrence present in the field. Phenomena represent a simulation element which highly depends on the simulation domain considered. As a consequence, the interface corresponding to this element *SimulationPhenomenon* is barely defined. As it can be observed in Figure 4.10 it includes two methods, which are *die()* and *mayBeSensedBy()*. The first method indicates the disappearance of the phenomenon from the field, and the second one identifies the kind of sensor that can perceive the phenomenon considered.

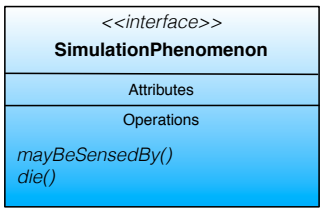


Figure 4.10: *SimulationPhenomenon* interface outline [Pujol-Gonzalez, 2008].

4.4.3 Agent

The key simulation element of RepastSNS is the agent, as this platform was conceived with the purpose of modelling a WSN from a MAS perspective. Figure 4.11 shows the structure of the *Simulation Agent* interface corresponding to this element.

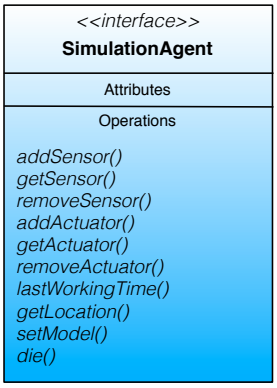


Figure 4.11: *SimulationAgent* interface outline [Pujol-Gonzalez, 2008].

As it was already presented in Figure 4.1, the agent consists in turn of different elements such as the CPU, the battery, sensors, actuators and communication interfaces. Although it is composed of these different elements, the *SimulationAgent* interface actually contains methods modelling the CPU behaviour (information processing and decision making). It also contains methods to manage the rest of components, such as *addSensor()* or *removeActuator()*. In the same way as the field informs the system about the appearance and dissapearance of phenomena through the sending of appropriate *SimulationEvents*, agents inform about the components that they have available (addition or removal of sensors and actuators). To perform this task, agents use the *SimulationEvents* shown in Figure 4.12 indicating sensors/actuators addition or removal.

The agent is a phenomenon present in the field, therefore it has a position in it. This position can be known through the method *getLocation()* of the interface.

Finally, the last method of the interface is the *lastWorkingTime()*. The function of this method was already introduced in Section 4.3.1. The platform's scheduler processes actions sequentially and when it has to deliver a *SimulationEvent* to a target object, it schedules this action at the moment this target object is available. An agent can tell the time it needs to process an event through this *lastWorkingTime()* method. The basic implementation of this method given by the platform evaluates this time as actual CPU time. However, its functioning can be adapted to any particular domain by overriding the method in a corresponding subclass. This method allows the scheduler to check the availability of an agent before delivering a *SimulationEvent* and consequently, delaying it if

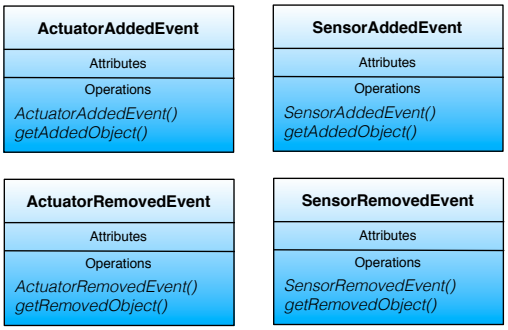


Figure 4.12: Notification events about sensor/actuator addition and removal [Pujol-Gonzalez, 2008].

necessary.

4.4.4 Sensors

Sensors are the elements of an agent through which it captures information about phenomena happening in the environment. As it can be observed in Figure 4.13, the *SimulationSensor* interface defines the perception capacity of a sensor relying on a filter element.

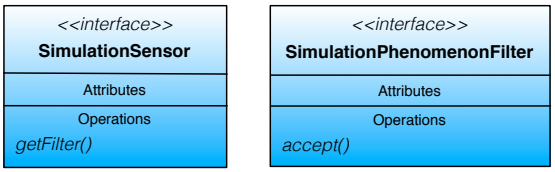


Figure 4.13: *SimulationSensor* and *SimulationPhenomenonFilter* interfaces outline [Pujol-Gonzalez, 2008].

Filters are specified by the *SimulationPhenomenonFilter* interface. The purpose of a filter is quite obvious: to distinguish phenomena observable by the sensor from those that cannot be perceived.

Sensors can be of two types: continuous or discrete. RepastSNS presents two different interfaces and their corresponding abstract classes for each kind of sensor, resulting then in the *DiscreteSensor* and *ContinuousSensor* interfaces and the abstract classes *AbstractDiscreteSensor* and *AbstractContinuousSensor*. Figure 4.14 depicts the schemes of the interfaces corresponding to these two kind of sensors.

- *Discrete sensors*: this kind of sensors collect information from the environment at particular moments in time. They can do it periodically or when they are specifically told to do it. The sensor action is triggered by

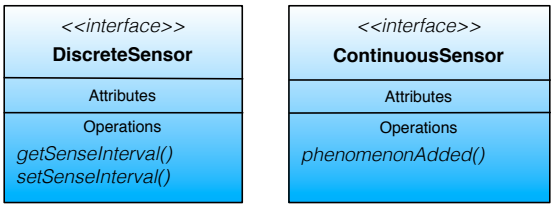


Figure 4.14: *DiscreteSensor* and *ContinuousSensor* interfaces outline [Pujol-Gonzalez, 2008].

the reception of a *SenseEvent*. The corresponding method to process this *SenseEvent* is contained in the *AbstractDiscreteSensor* class. This method executes in turn the *SensePhenomena()* method of the field which returns a list of all the active phenomena perceivable by the sensor and provides it to the *sense()* method of the sensor which captures it indeed.

- *Continuous sensors*: as their name say, continuous sensors receive information from the environment continuously. Therefore, their functioning scheme differs from the one associated to discrete sensors. To observe a phenomenon, the corresponding sensor abstract class (*AbstractContinuousSensor*) subscribes itself to the *SimulationField*. This subscription is established through calling the field's method *addNewPhenomenonListener()*. From this moment onwards, the continuous sensor is notified every time a new perceivable phenomenon appears in the field. The field informs the continuous sensor about the appearance of an observable phenomenon through calling the continuous sensor's method *phenomenonAdded()*. Once phenomena are detected, sensors can access their public information.

4.4.5 Actuators

Another element composing the agent is the actuator. Actuators can introduce new phenomena in the field and affect existing elements in the simulation. These elements only work discretely, hence they can act periodically or when they are told to do it. The *ActuateEvent* constitutes the actuation command, which is processed at the *AbstractSimulationActuator* class.

Interfaces corresponding to these two elements are extremely simple. None of them presents specific methods associated to particular functions. In fact, the *ActuateEvent* interface is empty, whereas the *SimulationActuator* interface only contains general methods for relating the element to the agent and other node's components.

4.4.6 Battery and Energy Consumption model

Most nodes in a WSN have a limited power supply provided by a battery. Hence, the aim of a WSN is not only to appropriately perform its monitoring task, but

to also do it maximising the life span of the network.

RepastSNS presents a basic model of the component battery of an agent (see Figure 4.15). It also considers and models the relationships established among simulation elements due to the energy consumption process.

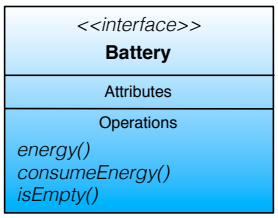


Figure 4.15: *Battery* interface outline [Pujol-Gonzalez, 2008].

Classes included in RepastSNS to model a battery component are the *Battery* interface and the *AbstractSimulationBattery* class. This component controls and manages the available energy of the node. Besides these classes, the simulation platform also presents a *InfiniteBattery* class extending from the previously presented abstract class. This *InfiniteBattery* allows modelling situation in which nodes are connected to an inexhaustible power supply. On the other hand, *DefaultBattery* class models a simple battery with an initial power capacity that diminishes as other simulation elements consume its energy.

Those simulation elements consuming energy from a node's battery are the components of the agent, that is, CPU, sensors and actuators. The energy consumption capacity of these elements is modelled through the implementation of the *EnergyConsumer* interface. This interface, as it can be seen in Figure 4.16, includes the method *hasEnergy()*, which tests the availability of energy for the component to do a task.

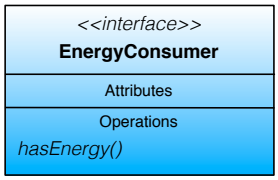


Figure 4.16: *EnergyConsumer* interface outline [Pujol-Gonzalez, 2008].

The energy consumption management is a complex process that depends on the characteristics of the particular kind of component. The energy consumption management of actuators is one of the simplest ones. Actuators are discrete devices that only consume energy at the time they are performing an action. In case there is not enough energy for this action to be made, it cannot be done. Sensors, on the contrary, consume energy just for being on. The platform discretises this consumption to avoid falsifying the battery level in time. Finally, the agent (CPU) consumption depends on its use. RepastSNS models

the consumption associated to this element from its standby and working intervals, consuming the energy corresponding to the previous period when changing state.

As it was already said, sensors and actuators consume energy from the agent's battery. However, the agent can turn off these components connected to its battery when they are no longer needed. The methods the agent use for turning on and off its components and to manage its own consumption can be seen in Figure 4.17 showing the skeleton of the *AbstractSimulationAgent* class.

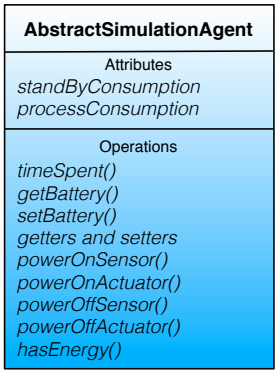


Figure 4.17: *AbstractSimulationAgent* class outline [Pujol-Gonzalez, 2008].

Finally, the ScheduleSNS also takes into account the energy management presented by checking the energy availability of a component before delivering an event.

4.4.7 Communication module

The communication module is the part of the platform which is in charge of simulating and managing the communication among agents. This module is barely defined and presents a basic definition of the interfaces corresponding to the elements involved in a communication process.

Agents communicate by sending and receiving data packets. This data have to be defined according to a *Data* interface. This interface is initially empty and can be adapted to the programmer requirements. For two agents to be able to communicate, agents transmit a special kind of phenomenon that can also be perceived by a particular kind of sensor. RepastSNS presents three interfaces defining the basic components involved in a communication: a transmitter, a receiver and a network interface (see Figure 4.18).

The *SimulationTransmitter* interface is associated to a particular kind of actuator able to transmit *Data* objects. The transmission of data is done through the creation of a particular phenomenon in the field. The *SimulationReceiver* is associated to a sensor able to capture this data. Finally, this *SimulationReceiver* and *SimulationTransmitter* are associated to a *SimulationNetworkInter-*

face, through which they can be handled.

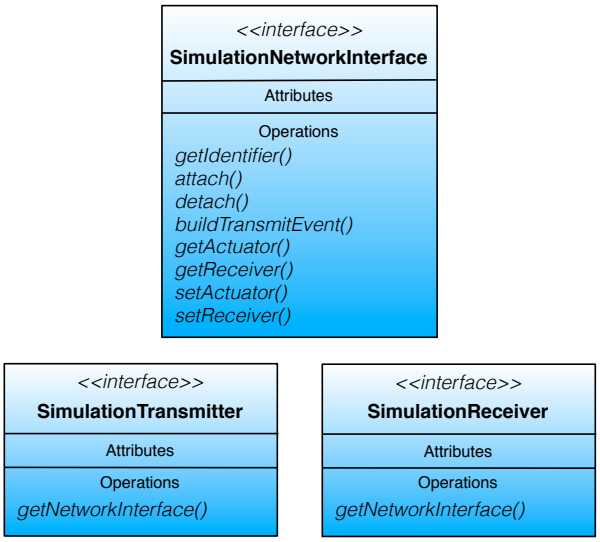


Figure 4.18: Communication elements interfaces outline [Pujol-Gonzalez, 2008].

Besides this, RepastSNS also presents a very basic *Wireless* module to implement radio communication. This module considers the particular phenomena transmitted to be *WirelessWave* generated by a *WirelessTransmitter* and perceivable by a *WirelessReceiver*. All these elements relate through a *WirelessCommunicationInterface* with the agent of which they are part. Each of these elements also implements the corresponding communication interface depicted in Figure 4.18. The *WirelessWave* emitted is the element that contains the information, that is, the *Data*.

The last communication layer that the platform provides corresponds to the *radial* set of elements. It offers a simple but effective view of wireless communication. The additional feature that this layer introduces refers to the nature of the wave. The *RadialWave* class defines a circular phenomenon that extends from its origin until a maximum distance, the communication radius.

4.4.8 Report

The report component represents the way to obtain information from the simulation provided by RepastSNS. Its functioning is based on the fact that every action occurred in the platform goes through the scheduler. Therefore, a copy of each event can be sent to the report component to register it, save it or analyse it correspondingly. This element is again only defined by an interface, *SimulationReport*. This interface contains the declaration of basic methods for its functionality implementation. As shown in Figure 4.19, one method relates the report to the model class and the other one allows it to capture events.

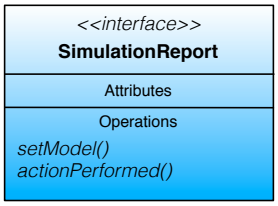


Figure 4.19: *SimulationReport* interface outline.

4.5 Conclusions

To study a Multiagent System and to analyse its behaviour when it implements a new algorithm require an important effort. The decision on which simulation platform to use may lead to increase that effort or to ease the task.

After studying and characterising some of the existing simulation platforms, and taking into account our demands, we decided to test COSA using a novel platform, RepastSNS. It represents a new platform but it runs over a well-known MAS simulation engine. This decision has posed us in a beta-tester position. Thereby, we have made different improvements and repaired some functionalities. For instance, we have fixed the scheduler work, managed basic functionality of sensors, corrected some methods definition, improved visibility problems of some classes and contributed to an easier debugging methodology by connecting elements that form part of the same agent. This effort, together with the characteristics that inspired RepastSNS definition, make it an appropriate simulation platform that meets our purpose.

RepastSNS has been conceived to study Sensor Networks from a MAS point of view. The platform, which is based on discrete events, provides a simulation structure and a set of components that ease modelling tasks of a sensor network as a MAS. Moreover, this component-based definition contributes to make it extensible. Components in the platform relate to each other through message exchange what favours scalability. The establishment of new connections or the addition of components just changes the way existing elements relate but do not affect them. Typical components provided by the platform are agents, sensors and actuators. All they provide common functionalities offered by a sensor network, what eases application development tasks, but at the same time, they still keep a high abstraction level, that make them general and valid for different application domains. Finally, the simulation time control provided by the scheduler allows to fulfil the initial requirements desired for the simulation environment. All these reasons has led us to select RepastSNS as the base engine to perform our experiments.

Chapter 5

Energy and Communication Aware WSN

An attractive feature of RepastSNS is its general character. Although it has been designed to model Sensor Networks as Multiagent Systems, it does not make any assumption about the kind of environment or network to be modelled. This characteristic demands additional implementation effort to model not only the agents' behaviour, but also basic attributes referred to the node regular working scheme, such as energy consumption, for instance.

Therefore, before tackling the Multiagent System algorithm development, we need to design and plan the network and the environment model that we are going to implement. In this chapter, we present the approach selected, and the layers modules established for this purpose. The first layer built upon RepastSNS focus on the consumption and communication capabilities of the nodes. The details of this layer's definition are given in this chapter.

5.1 Application development structure

RepastSNS lies over Repast Multiagent Systems simulator to contribute its basic sensor network definition. Proper implementation of the MAS algorithm proposed in Chapter 3 on the platform requires the definition of the environment to be monitored and the network that performs this surveillance task. The goal of COSA is to provide the agents with a behavioural policy that allows them to perform their mission correctly while they use their resources efficiently. The physical devices holding the agents are subject to energy and communication constraints. This situation motivates the interest in energy saving. As a consequence, to adequately evaluate this MAS algorithm, we also need to model these conditions referred to energy and communication capabilities. The approach selected for the application development can be observed in Figure 7.1.

According to the conception principles of RepastSNS, we propose a modular and growing structure. The work to be developed from this point onwards

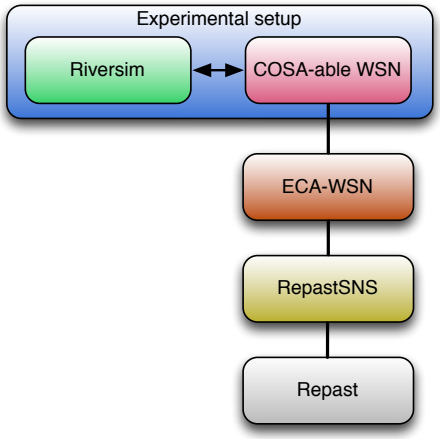


Figure 5.1: Development Structure.

focus on the upper levels of the diagram. We distinguish three interconnected modules that are grounded in RepastSNS and Repast. The basic one is *Energy and Communication Aware Wireless Sensor Network* (ECA-WSN) and over it, we include the *COSA-able Wireless Sensor Network* (COSA-able WSN) module. *River simulator* (Riversim) module is grouped together with COSA-able WSN by the *Experimental setup*.

The ECA-WSN module broadens, and specifies at the same time, the communication capabilities of the nodes composing the network. This module defines the way agents can perform primary actions, such as sampling and communicating. The model selected to represent these actions conditions the tasks and work to implement this module.

Tasks dedicated to the proper implementation of the intelligence provided by COSA in the agents belong to the COSA-able WSN module. The work developed within this frame aims at creating the intelligent agents' structure and supply them with the different tools that they may need.

Differently from the previously described modules, the Riversim do not pay attention to the agents or nodes' capabilities. It focuses on the environment, its model and definition.

Finally, the Experimental setup frame aggregates these two last modules, COSA-able WSN and Riversim. The first one defines the intelligence of a set of nodes, whose actuation capabilities have already been defined; whereas the second module sets the environment holding those agents and where they perform their activities. The Experimental setup defines the relationship between them. This module finally builds the target application itself and enables its study and analysis (the original purpose of this work).

In the following sections, we present the model selected to manage the energy consumption associated to the agent. The communication module developed, and the characteristics added are also specified later. These two facets of the

agent have been implemented to enrich the previous models and sustain higher level applications.

5.2 Energy management model

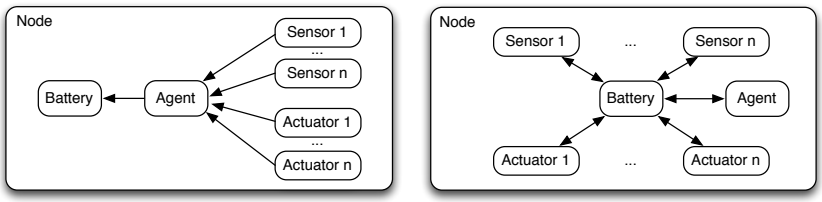
All the elements that compose an agent demand energy to perform their corresponding tasks. The energy consumption represents a fundamental facet of an agent's functionality. The model selected to deal with this aspect affects the level of complexity of the experiments to perform.

As described in Section 4.4.6, RepastSNS offers a basic model of this functionality. RepastSNS platform defines a *Battery* element connected to an agent. All the elements of the agent that may need energy at a certain moment are considered as *EnergyConsumers*. According to an *EnergyConsumer* operation, these elements could check the state of the *Battery*. Nonetheless, when one of these *EnergyConsumer* elements requires a certain amount of energy, it tells the agent about this need, which in turn, connects to the *Battery*. In case the *Battery* can supply the needs of that element (sensor, for instance), the energy is spent, and the action executed. Figure 5.2(a) shows a schematic representation of the communication with the battery.

The approach provided by RepastSNS to tackle the continuous consumption is also quite simplistic. On the one hand, energy consumption associated to Continuous Sensors is discretised in fixed intervals. This strategy obliges the programmer to establish the period for continuous consumption update beforehand. On the other hand, CPU's continuous consumption is considered when this element changes its state. Taking into account that the agent acts as the CPU, and it manages all the other components that form part of the node, it may change from idle to active quite frequently. As a consequence, considering the consumption associated to the idle state at these times seems a good option for this particular element.

ECA-WSN defines an energy consumption management policy that lies on the same essential points as RepastSNS approach: the distinction between the energy supplier and consumer elements, and also, between continuous and sporadic consumption. Apart from this, the viewpoint selected to implement and develop this consumption management strategy is very different.

The battery considered within this ECA-WSN module is capable of self-management. According to this model, the battery knows all the elements that can connect to it to demand energy and it can also handle these demands. Hence, from this perspective, the battery is not a passive element to which the rest of the node's components connects to demand service (energy provision). Moreover, consumer elements connect to their energy supplier directly, that is, the battery (see Figure 5.2(b)). The agent is not in charge of managing the energy demands of its components. When an actuator, for instance, needs to consume energy, it can demand it right to its associated battery. The change of how elements relate compared to the approach given by RepastSNS introduces a significative improvement regarding, especially, to continuous consumption man-



(a) RepastSNS model. (b) *ECA-WSN* model.

Figure 5.2: Energy query direction.

agement. Among the positive features provided by this policy, we can mention that it avoids burdening the agent with continuous demands of energy from other elements. Besides this, the self-management capacity of the battery eases the functioning of those energy consumer elements with standby consumption, as the battery itself can demand an update of the standby consumption whenever it finds it appropriate. Thereafter, this battery's feature avoids a misleading standby energy management associated to elements' state change, and it also releases energy consumers from making periodic demands of standby energy. The consumption policy definition is also simplified as it does not require prefixing consumption periods and energy demands beforehand.

Proper implementation of this model for energy consumption management is presented next. We present the implementation developed for the battery and the elements demanding energy separately.

5.2.1 Common features

Energy management represents a fundamental function of a WSN. Therefore, selecting an appropriate implementation strategy is crucial to help in later development or debugging tasks.

To implement the model described above, we decide to keep the approach used in RepastSNS. That is, elements are specified by an interface and an abstract class. This decision favours the development of a scalable and extensible ECA-WSN module, which is also coherent with its lower levels.

On the one hand, elements' interfaces just outline the characteristics that an element is going to have. On the other hand, an abstract class provides with a basic definition of the methods required by that element.

We introduce the interfaces associated to battery and consumer elements in this section. A detailed description of each of them can be found in its corresponding subsection. However, we present them here together to allow for a more coherent explanation of our work.

The structure of the *Battery* interface can be observed in Figure 5.3. It contains the declaration of the methods *energy()*, *consumeEnergy()* and *isEmpty()*.

The interface defined for consumer elements is the *EnergyConsumer* interface (see Figure 5.3). This interface, associated to elements capable of demanding en-

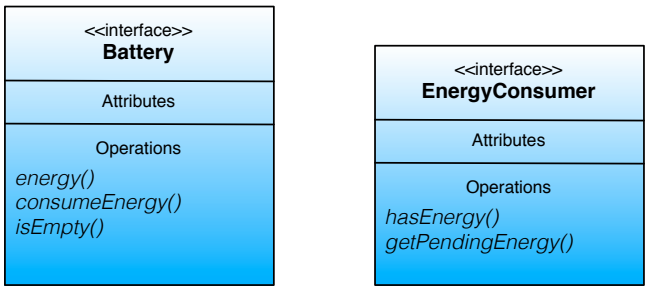


Figure 5.3: *Battery* and *EnergyConsumer* interfaces.

ergy to the battery, includes the methods *hasEnergy()* and *getPendingEnergy()*. As it name states, the *getPendingEnergy()* method returns the amount of energy consumed by the element since the last time it demanded energy to the battery. The implementation of this last method depends on the kind of element being modelled, but its meaning does not change.

If we compare these interfaces to its counterparts provided by RepastSNS, we realise that the difference between them resides in the *EnergyConsumer* interface, that now incorporates the *getPendingEnergy()* method. Hence, this method plays a key role in the implementation of the energy consumption policy of ECA-WSN.

5.2.2 Battery

The battery of the nodes represents a central concept of an energy model. Its associated interface is shown in Figure 5.3. The methods contained respond to basic functionalities, such as the evaluation of the current capacity of the battery, the consumption of a certain amount of energy and the verification of the emptiness condition.

The abstract class *AbstractBattery* presents a basic implementation of each of these methods. However, to provide a battery with a certain level of self-management and to make it able to handle energy demands, it needs to know which are the elements capable of demanding energy to it. Thereafter and to satisfy this need, the *AbstractBattery* class maintains a list of all the elements implementing the *EnergyConsumer* interface which are associated to it. This list is represented by the *consumers* attribute in Figure 5.4.

The energy management policy developed for ECA-WSN leads the battery to be able to update and know its capacity whenever a consumer demands energy. Instead of receiving periodic or sporadic standby energy demands from consumers, the battery can ask them how much standby energy they have consumed since their last demand whenever it finds it appropriate. It simply needs to call the consumers' method *getPendingEnergy()*. Consequently, the correct development of this task demands an accurate management of the consumers' list. That is, this list must keep track of the addition and removal of consumers.

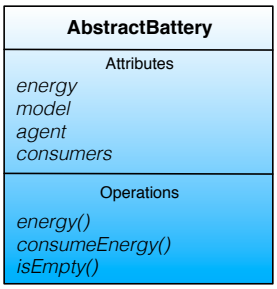


Figure 5.4: *AbstractBattery* class outline.

The reception of *SimulationEvents* referred to the on/off switching of consumers changes the relationship of these elements to the battery, but it also updates the battery's perception about them. This way, as it was desired, both the battery and the consuming elements know each other at any time.

The implementation of the methods inherited from the *Battery* interface, *isEmpty()* and *consumeEnergy()* are conditioned by the self-management capacity of the battery. The *isEmpty()* method shows the utility of keeping a list of consumers. Every time that the battery is asked whether it is empty or not and before checking this condition, it updates its capacity to the right value at that time. In order to do this, the battery asks each of its consumers the amount of standby energy consumed in their latest inactivity period and subtracts these values from the current energy level. This simple process allows the battery to keep its capacity value up-to-date and, consequently, to accept or discard actions based on this updated information. Figure 5.5 depicts a flowchart of the actions described.

The functioning scheme of the *consumeEnergy()* method also adjusts to this informative perspective. Hence, before subtracting the demanded quantity of energy to the capacity level of the battery, it calls its own *isEmpty()* method to update its capacity value. The desired consumption action will be done or not depending on the result of this update and the quantity of energy demanded. Thereafter, the battery itself is in charge of managing other elements' accesses to it and can keep its energy level updated whenever it is required.

5.2.3 Energy consumers

Sensors, actuators and the agent itself constituting a node require energy to perform their tasks or just to be on. All these elements consume the energy available in the battery that, together with all of them, form a node of the network.

The basic behaviour of a battery has been already presented in the previous section. The description of how energy consumers adapt to an energy consumption management in which the battery can handle demands and consumers can ask for energy directly to the battery follows next.

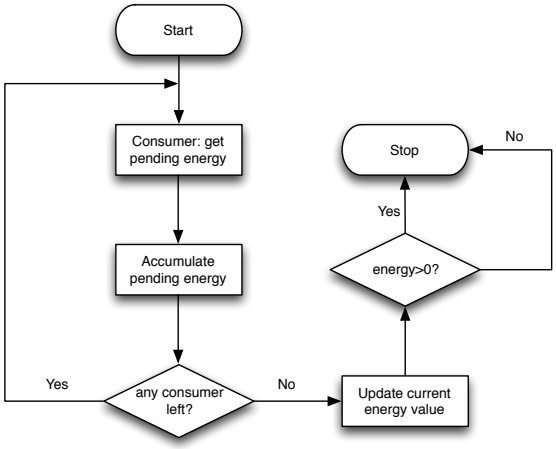


Figure 5.5: Flowchart corresponding to the *isEmpty()* method.

The definition of the behaviour of an energy consumer element relies on the *EnergyConsumer* interface (Figure 5.3). The specification of the methods that this interface contains cannot be done in a unique abstract class common to all consumers. Each consumer behaves differently according to its nature, for instance, the energy demands of a continuous sensor are not equivalent to those of an actuator's action. This fact prevents us from the creation of an abstract consumer class. The introduction of the energy management policy in the consumer elements is done through the corresponding abstract classes of each element (sensors, actuators and agent). These abstract classes implement the *EnergyConsumer* interface and define basic functions coherent with the energy policy developed in this module.

To describe the specification of the *EnergyConsumer* interface methods, we focus on each element and show how the element's nature condition their implementation.

Sensors

To allow a sensor to demand and consume energy it has to implement the *EnergyConsumer* interface. The definition of a sensor, as any other element in the system, is based on a set of classes and interfaces. Figure 5.6 shows the primary part of this hierarchical structure. One of the classes that defines a sensor must implement the aforementioned interface.

Energy consumption function is common to every kind of sensor. Therefore, it seems reasonable that the most generic sensor class implements the *EnergyConsumer* interface. This decision leads to the redefinition of the *AbstractSimulationSensor*.

The *AbstractSimulationSensor* class implements the *EnergyConsumer* interface described in Section 5.2.1. Nonetheless, in this class only the *hasEnergy()*

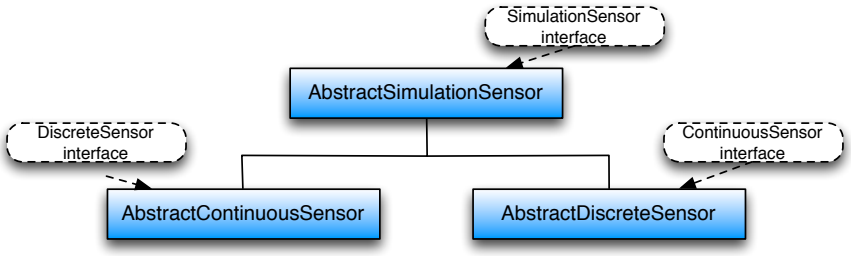


Figure 5.6: Structure of classes for sensor definition.

method is specified. The aim of this method is to test the availability of energy in the battery.

The energy management policy implemented in ECA-WSN allows for direct connection between the sensor and battery, therefore, the action performed by the *hasEnergy()* method is quite straightforward. First, the existence of a battery is verified to ask immediately about its state by calling the battery's *isEmpty()* method.

The *hasEnergy()* method allows the consumer (a sensor in this case) to ask the battery about its emptiness condition. The *getPendingEnergy()* method plays the dual role, as the battery can invoke this consumer's method to know the cumulative energy demand. The value of this cumulative demand depends on the sort of sensor, therefore, the implementation of the *getPendingEnergy()* method is done at a more specific class in the sensor hierarchical structure definition. These classes, particularly *AbstractDiscreteSensor* and *AbstractContinuousSensor*, specify the behaviour of the sensor, what favours an appropriate implementation of this method.

- Discrete Sensors

Sensors in the system can be continuous or discrete. Discrete sensors are specified by the *DiscreteSensor* interface and the *AbstractDiscreteSensor* class. The basic functioning of a discrete sensor is simpler than a continuous one. Figure 5.7 outlines the methods and attributes contained in the *AbstractDiscreteSensor* class. Discrete sensors only demand energy at particular instants of time. Particularly the consumption per action is a fixed quantity requested when the agent samples the environment. The definition of the *getPendingEnergy()* method is trivial then. This method just always returns zero, as standby consumption is negligible.

- Continuous Sensors

Unlike discrete sensors, continuous sensors are continuously perceiving the environment; therefore, their corresponding *AbstractContinuousSensor* class requires proper implementation of the *getPendingEnergy()* method to reflect this fact.

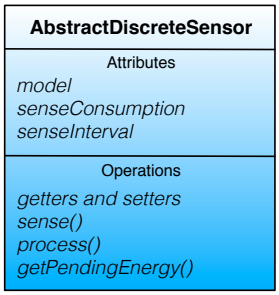


Figure 5.7: *AbstractDiscreteSensor* class outline.

The battery is the element in charge of updating an element's standby consumption. Whenever the battery requires to know its exact capacity, it executes its own *isEmpty()* method, which in turn calls the *getPendingEnergy()* method of each associated consumer. Consequently, and in this particular case, the energy consumed by a continuous sensor is then discretised at irregular intervals. In order to do this, each continuous sensor has to keep track of the last time it consumed energy. The *AbstractContinuousSensor* class incorporates a property to meet this purpose and save this timestamp. The *lastConsumptionTime* variable updates its value every time the sensor performs a sampling action and when the *getPendingEnergy()* method is called by the battery. The functionality of the *getPendingEnergy()* method is quite obvious now. This method evaluates the standby energy consumed since the time instant marked by the *lastConsumptionTime* variable and until the current time, and returns this value. This new property can be observed in Figure 5.8, which shows this class' primary methods and attributes.

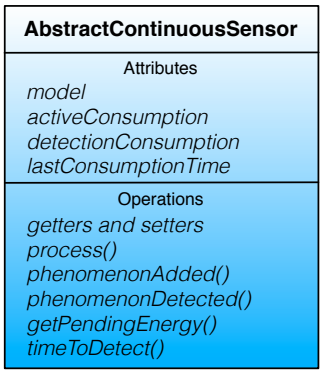


Figure 5.8: *AbstractContinuousSensor* class outline.

The energy management of the proper action performed by a continuous sensor is also tackled by the *AbstractContinuousSensor* class. Although continuous

sensors continuously perceive the environment, this does not imply that whenever an observable phenomenon appears in the field it will be detected. Sensors may or may not have enough energy to perceive the phenomena appearing in the environment. To consider this circumstance, we distinguish two methods: *phenomenonAdded()* and *phenomenonDetected()*. The *phenomenonAdded()* method reflects the appearance of an observable phenomenon in the field, whereas the *phenomenonDetected()* method will deal with the phenomenon processing.

The execution of *phenomenonDetected()* by the sensor to get information from the environment is subject to the successful execution of the *phenomenonAdded()* method. This method, *phenomenonAdded()*, evaluates the amount of energy required by the sensor to actually perceive the phenomenon. If the battery can provide this energy to the sensor, then *phenomenonDetected()* method is executed for further processing the collected data.

The evaluation of the energy needed by the sensor to perform a sampling action is based on the time required to perform the action, assuming that the device consumption's specifications are expressed per time unit. To take into account this last feature, we declare the *timeToDetect()* method in the *AbstractContinuousSensor* class to return this value. Proper implementation of these methods is left to more specific classes referred to the actual sensor being modelled.

Actuators

Actuators rely on the *SimulationActuator* interface and the *AbstractSimulationActuator* class. Consistently with the sensor's structure, the suitable class to implement the *EnergyConsumer* interface is *AbstractSimulationActuator* (see Figure 5.9).

The behavioural model defined for actuators allows for the implementation of the two methods contained in the *EnergyConsumer* interface within this class. According to the energy management pattern established, the *hasEnergy()* method just tests the existence of a battery associated to the element and whether it is empty. Regarding the *getPendingEnergy()*, analogously to the *AbstractDiscreteSensor*, this method just returns zero. We neglect the actuator's standby consumption and assume that actuators only require energy when acting.

The energy management of the actuation is also incorporated to this class. The *getEnergyToGenerate()* method aims at evaluating the energy needed for the actuator to perform its action. At this level of the element definition, this method just informs about the actuator's energy consumption per time unit. The proper implementation of this method has to be done in a more specific class. The performance of an actuator is conditioned to the availability of energy to supply its needs.

Agents

Finally, the last element consuming energy from the battery is the agent governing the node. Again, the *AbstractSimulationAgent* class implements the *Energy-*

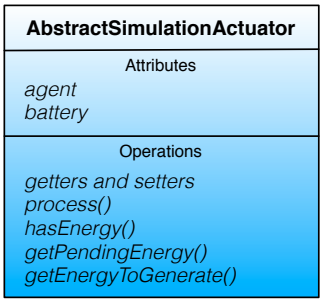


Figure 5.9: *AbstractSimulationActuator* class outline.

Consumer interface and its methods. On the one hand, the *hasEnergy()* method is exactly the same as its counterparts in the other classes already described. On the other hand, the *getPendingEnergy()* method conforms to the characteristics of this element. Specifically, it evaluates the standby consumption relying on the *lastWorkingTime* property of the agent. Based on it and on the current instant of time, this method evaluates the energy consumed. In the case of an agent, the evaluation of the consumption per action depends on the time invested in each particular action (as this consumption specification is also expressed per time unit). Figure 5.10 shows the main characteristics of the *AbstractSimulationAgent* class.

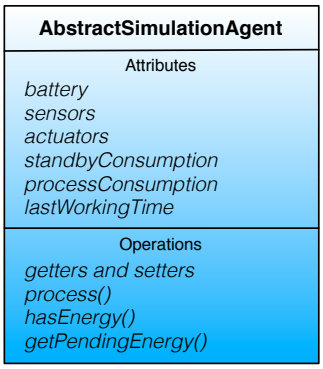


Figure 5.10: *AbstractSimulationAgent* class outline.

Hence, from the point of view of the battery, the energy consumption management of the agent, or a sensor or an actuator is equivalent. The standardisation provided by the energy management model implemented in ECA-WSN simplifies the individual elements operation in this respect. Concurrently, it takes into account the two possible consumptions considered in the model for every element, independently of whether it presents it or not. As a consequence, the energy model implemented renders a consistent system.

5.3 Communication model

In general terms, the election of a communication model for a MAS is equivalent to establishing the agents' capabilities of sending and receiving messages. As most MAS applications rely on the communication capacity of the nodes, this decision is of critical significance. A correct model of this feature that provides a solid and general communication structure favours the development of applications over the platform. ECA-WSN module introduces a novel communication model for the agents. The implemented model enriches the capacity of the agents without impairing the generality of the process.

RepastSNS provides with a complete but very general communication structure that misses some basic features of this function. Moreover, the energy model already incorporated to the ECA-WSN module affects communication elements, such as the transmitter and receiver. Hence, the initial definition of the communication module delivered by RepastSNS has been revised.

The communication capacity of the nodes has been considered from a generic point of view and, in this vein, primary communication features have been added. Nonetheless, these new features do not constrain the set of scenarios that can be simulated, rather the opposite, they may ease the development tasks associated to different applications. A new functionality incorporated is the capability of emitting directed and broadcast messages. Assuming the use of a *Radial Communication* model, as described in Section 4.4.7, broadcast messages can be received by any agent within the communication radius. On the other hand, directed messages are sent towards a specified recipient. The inclusion of this new feature implies, and requires, changes on the elements in charge of transmitting and receiving messages (transmitter and receiver).

To attain a more realistic modelling of these transmitter and receiver elements, as well as of their actions, the communication bandwidth associated to these devices is also taken into account. The introduction of this property not only allows for a more correct simulation, but it also offers the opportunity to make transmitter and receiver elements comply with the energy consumption policy adopted. Given the bandwidth assigned to a communication element and the size of the message to be transmitted or received, the amount of energy needed for the task completion can be easily calculated from regular specifications of communication devices. Therefore, the communication model completes and complements the work presented in the previous section.

In the following sections, it is described how the ECA-WSN module implements this model. The incorporation of the bandwidth and the directed communication ability requires the design and development of a new communication structure.

5.3.1 Data elements

The implementation of the communication model conforms to the *interface-abstract class* structure already used for the energy model. This approach guar-

antees the accomplishment of an uniform ECA-WSN module, which is also coherent with its lower levels (RepastSNS).

Agents communicate through message exchange. The basic characteristics of these messages are defined through the *Data* interface and the *AbstractData* class.

The *Data* interface declares basic methods associated to the message emitter identification. Besides this, it includes a *getBytes()* method to return the size of the message. This interface also contains an *enum* constant that includes the size (in bytes) of the different data types that the message may carry.

The *AbstractData* implements the *Data* interface. This class includes the *sender* property and the corresponding getter and setter methods. The *getBytes()* method defined in the interface also presents a basic definition in this class. In addition, *AbstractData* provides with a method that checks the identity of a the *data*'s sender. This method results very useful when the agent is engaged in conversations with its neighbours. Figure 5.11 outlines the definition of this class.

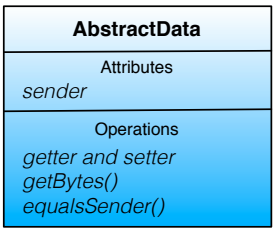


Figure 5.11: *AbstractData* class outline.

This modest structure provides all the basic elements needed to completely define exchangeable messages for the agents. Despite its simplicity, it is flexible enough to allow the sending of different kinds of data to one or more addressees.

5.3.2 Communication elements

The introduction of the bandwidth concept and the unicast message emission demands the creation of a new communication structure. This structure embraces the fundamental elements involved in communication, which are a transmitter and a receiver. Moreover, these elements are linked together by another element, a network interface. In the following we describe how these elements and their behaviour develop the desired communication model.

Network Interface

The network interface has been conceived as a binding element for the communication devices. This element is in charge of managing the transmitter and receiver and linking them to the agent managing the node.

In the same vein as all other elements of the ECA-WSN module, the network interface definition is based on an interface and an abstract class, which are *SimulationNetworkInterface* and *AbstractNetworkInterface* correspondingly.

The purpose of the *SimulationNetworkInterface* is to outline the function of the communication module. In order to do this, it declares methods to connect the transmitter, receiver and the agent. This interface is equivalent to the *SimulationNetworkInterface* presented in Section 4.4.7 but for the inclusion of an overloaded method: *buildTransmitDataEvent()*. This method is especially noteworthy, as this is the one that the agent invokes to tell the transmitter its intention of sending a message, wether it is a broadcast or a directed message. This method shows how the simulation interface element acts as both simultaneously, as a bond and a frontier between the agent and the communication elements in the node. This perspective favours the agents' abstraction from the working scheme of the communication element while still allowing their interaction. The implementation of the methods defined in this interface appears in the *AbstractNetworkInterface* class.

To fulfil the implementation of methods related to fixing the relationship between communication elements and the node is very straightforward. The definition of the *buildTransmitDataEvent()* method requires the support of a particular *SimulationEvents* structure. Figure 5.12 shows this structure of events.

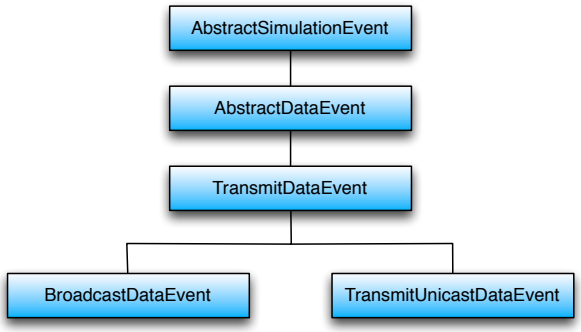


Figure 5.12: Transmission events structure.

The structure starts at the most basic event class, the *AbstractSimulationEvent*. From this class extends the *AbstractDataEvent*. The *AbstractDataEvent* incorporates a *Data* property and its associated management methods. Therefore and as it name indicates, this kind of event can contain a *Data* type message. *TransmitDataEvent* is an abstract class that implements the *ActuateEvent* interface. This feature allows this *TransmitDataEvent* to be recognisable by transmitter elements. Finally, from this last class, two instantiable classes extend, *BroadcastDataEvent* and *TransmitUnicastDataEvent*. These particular events are aimed at providing the communication modes indicated at their own names. *BroadcastDataEvent*, as its name states, is used for broadcast communication, and it does not include any novelty when compared to its superclass. However,

TransmitUnicastDataEvent adds a new property, the recipient. This event is responsible for initiating a directed communication. To comply with its mission, it does not only include the recipient identification, but also methods to deal with this property.

The addition of these classes broadens the actuation capability of transmitters. The reception of a *TransmitUnicastDataEvent* or a *BroadcastDataEvent* by a transmitter tells this element to actuate in either of the two ways. Depending on the kind of event received, the transmitter emits a message of the type specified by the event and with the parameters it gives.

Transmitter and Receiver

The simulation network links the transmitter and the receiver together and places them in the node's structure. Transmitter and receiver are different elements with dual functionalities. The transmitter is an actuator able to send messages, whereas the receiver is a continuous sensor capable of perceiving these messages and extracting the information that they may carry. This naïve analysis results very helpful to describe transmitter and receiver classes as part of the communication module. Moreover, this perspective favours the comprehension of how these elements integrate into the node structure and adapt to the energy management model of ECA-WSN module.

According to the communication model proposed for ECA-WSN module, transmitter and receiver elements are subject to a certain communication bandwidth. This concept is introduced in the corresponding elements' definition through interfaces and abstract classes.

Both the *SimulationTransmitter* and *SimulationReceiver* interfaces declare exactly the same couple of methods: *getNetworkInterface()* and *getBandwidth()*. The first one will link the element to a network interface, and the second one will allow to take into account the bandwidth property when operating. Despite its identical appearance, these interfaces are completely different. They correspond to the definition of different elements, and this divergence can be appreciated in their superclasses. If we recall the previous analysis, the *SimulationTransmitter* interface extends from *SimulationActuator*, whereas the *SimulationReceiver* interface extends from *ContinuousSensor*, already presented in Section 5.2.3.

The definition of these elements continues with the established protocol. Abstract classes implement these interfaces and define the methods declared in them. Hence, complying with this, the abstract class *AbstractTransmitter* implements the *SimulationTransmitter* interface (see Figure 5.13). This class includes the *Bandwidth* property and methods for its management. The analogous process applies to the *AbstractSimulationReceiver*. Nonetheless, the *AbstractTransmitter* declares a new method that does not appear in the *AbstractSimulationReceiver*. This method is the *timeToTransmit()*. As it name states, *timeToTransmit()* aims at computing the time required for the transmitter to emit a message. This time, together with the energy consumption per time unit associated to the transmitter, returns the energy cost of the transmission action. The *timeToTransmit()* method provides the same functionality offered by the

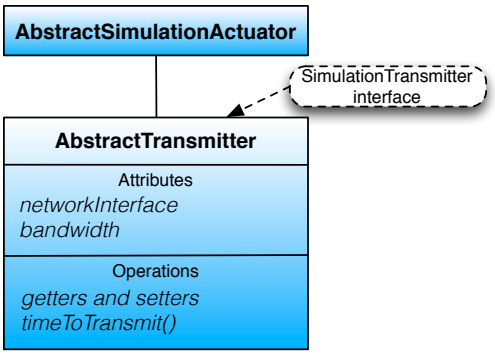


Figure 5.13: *AbstractTransmitter* class outline.

method *timeToDetect()* declared in the class *AbstractContinuousSensor*. Once again, the duality between these two elements becomes relevant. The *AbstractSimulationReceiver* extends from *AbstractContinuousSensor*, incorporating then this functionality (see Figure 5.14). However, the *AbstractTransmitter* extends from the *AbstractSimulationActuator*, which did not present it.

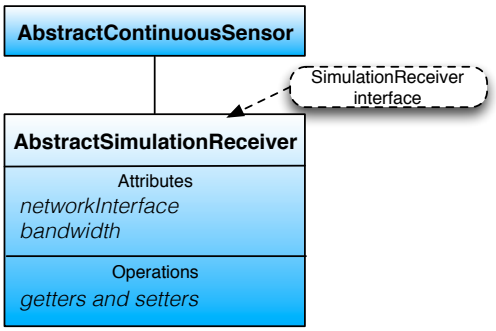


Figure 5.14: *AbstractSimulationReceiver* class outline.

This brief review of the elements definition and dependences situates them in the node structure. It also shows the interdependence existing between the energy and communication model defined for ECA-WSN module.

Wireless Communication Instantiation

Communication is a key cornerstone of both, MAS and WSN. In the previous sections, the communication model, and its integration in the simulation elements, have been described. The work performed until this point provides with a very general simulation environment that is not constrained to any particular communication technology. However, as our aim is to test the behaviour of the COSA over a WSN, the ECA-WSN module includes a set of classes defin-

ing proper communication devices. This step ahead favours the development of upper MAS application of any kind, but that require the support of a WSN.

RepastSNS already presented a group of classes giving this functionality in a very broad sense. ECA-WSN enriches that application by adopting the energy and communication model proposed for this module.

The classes that specify this communication capability are *WirelessWave*, *WirelessReceiver*, *WirelessReceiverFilter* and *WirelessTransmitter*. The implementation of these classes allows for better comprehension of the models exposed and their working scheme.

The *WirelessWave* represents the phenomenon emitted by the *WirelessTransmitter*. Its nature is not affected by the communication model of the ECA-WSN module and it just contains the message to transmit.

The *WirelessReceiver* represents, as its name states, the receiver device. This class extends from the *AbstractSimulationReceiver* and constitutes the appropriate frame to define methods specific for this type of communication. The *phenomenonDetected()* method inherited from *AbstractContinuousSensor* class can be adapted to the *WirelessWave* phenomenon and extract the contained message. Similarly, the *timeToDetect()* method can define how long it takes to detect the wave at this level of implementation. To return this duration, the length of the message and the bandwidth of the receiver are considered. This result, together with the consumption specification per unit time, permit to calculate the energy consumption involved by the action.

The detection of a *WirelessWave* by a *WirelessReceiver* demands the use of a *WirelessReceiverFilter*. The function of this element is refined to discard messages not addressed to the node and emitted by the node itself. The introduction of unicast communication in the ECA-WSN module allows the agent to avoid the process of messages not addressed at them.

Finally, the *WirelessTransmitter* is the element that generates the event that triggers the transmission action. It is, therefore, in charge of transmitting directed and broadcast messages and generating the corresponding waves carrying those messages. As it happened with the *WirelessReceiver*, this class is at the specification level of the transmitter definition that enables the evaluation of the time required to transmit a message and the energy incurred in this task. That is, the *timeToTransmit()* method declared in the *AbstractTransmitter* class, and the *getEnergyToGenerate()* method inherited from the *AbstractSimulationActuator* class are properly implemented in this class. The definition of these methods connects the two facets of the ECA-WSN module, the energy and communication models proposed collide and fulfil the element definition. Any other communication model lying on the same principles can worked over the ECA-WSN module. In fact, the *Radial* communication module proposed in Section 4.4.7 can run over it just by extending from these classes.

The set of classes described represents a complete communication module that includes a basic characterisation of this function and models different kind of communications. The implementation of this set of classes finishes the definition of the ECA-WSN module. This work emphasises how the energy and

communication models interrelate and come full circle.

5.4 Conclusions

The ECA-WSN module constitutes an elemental layer for later application development. It defines generic and simple communication capabilities while carefully fixing the relationship between the elements involved in the process. The energy consumption associated to communication actions is evaluated according to the general management policy proposed for this module. This energy management policy uniforms the energy consumption treatment for every element. Thereafter, this module represents an intermediate layer between the application and RepastSNS, that enriches and standardises the use of the network structure provided by the last one.

Chapter 6

Coalition Oriented Sensing Algorithm based WSN

The completion of the ECA-WSN module prepares RepastSNS platform for the implementation of the Multiagent Systems algorithm defined in Chapter 3. The COSA-able WSN module gathers together all the elements especially conceived to allow the agent to behave according to COSA. Thus, in this chapter we present not only the model of the agent implementing COSA, but also other elements supporting the agent's intelligent behaviour. These elements range from the special kind of battery designed to meet COSA features to the mathematical dimension of the agent. The structure created in this module for proper COSA implementation into an agent favours a better understanding of the proposed algorithm.

6.1 Power Supply

The battery of a node is the element in charge of providing energy to all the other node's components to perform their functions.

Agents implementing COSA can make an intelligent use of their resources, among them, their energy. One particular feature of this intelligent management is saving a small quantity of energy before reaching complete depletion. This so-called *Energy security level*, which was introduced in Chapter 3, is preserved by the agent to be able to send a last disconnection message. The class providing this functionality is the *CfAbstractBattery*.

CfAbstractBattery class extends from *AbstractBattery*. This class contains two properties as it can be observed in Figure 6.1. The *InitialEnergy*, that represents the maximum capacity of the battery, and the *deathThreshold* property. This last property represents the amount of energy that the battery keeps for the node's last action (the equivalent E_{sl} value appearing in Formula 3.3). The values set to these properties are specified when the object is created through its constructor method.

The existence of an energy threshold determines two different kinds of consumption: energy consumption above and below the threshold. Battery energy above the *deathThreshold* can be invested in whichever regular task of the node, like sampling, processing or communication. The energy remaining below this threshold cannot be used for any of these purposes. This distinction leads to the redefinition of the emptiness concept and the split of the methods inherited from *AbstractBattery* aimed at consuming energy and checking its availability.

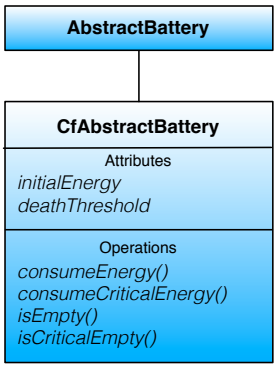


Figure 6.1: *CfAbstractBattery* class outline.

Therefore, energy management in this class is done through four different methods: *consumeEnergy()*, *consumeCriticalEnergy()*, *isEmpty()* and *isCriticalEmpty()*. The aim of the first two methods is to decrement the battery's energy by the corresponding quantity demanded. The *isEmpty()* and *isCriticalEmpty()* methods check, as their names state, the availability of energy.

The *isEmpty()* method overwrites the method inherited from *AbstractBattery*. In this case, the comparison condition for emptiness is set to *deathThreshold* value instead of zero. The evaluation of the battery current energy value is performed from *EnergyConsumers* as it was done in the *AbstractBattery* class. In these conditions, a node considers that it has an empty battery unable to provide energy for any regular activity when the energy level reaches the *deathThreshold* value. If this condition holds, an inner communication event triggers. An event of *DeathEvent* type is immediately sent by the battery to the agent to initiate the last disconnection message emission. On the other hand, the *isCriticalEmpty()* method just checks the emptiness condition normally with a zero value.

The *consumeEnergy()* method just subtracts the demanded energy as long as there is enough available energy. The *consumeCriticalEnergy()* mimics the behaviour of *consumeEnergy()* but checking the battery condition through the *isCriticalEmpty()* method. The only activity able to trigger the consumption of this critical energy is the disconnection message sending.

This class allows for the energy management of a battery element according to COSA. The design of a battery for COSA nodes is inspired by a real device, the WaspMote [Libelium, 2012b]. Therefore, we refer to its technical characteristics

to define our simulation component.

The *WaspMoteBattery* class extends from the *CfAbstractBattery*, and it is quite simple. It only contains an energy property that indicates the maximum capacity of the battery. This class presents two overloaded constructor methods that can be distinguished by the admission of a parameter or not. This parameter represents the *deathThreshold* value. The existence of two constructor methods allows for a general use of this battery class and not only for COSA nodes.

The constructor taking no parameters assumes a zero value for the *deathThreshold*. Hence, it creates a regular battery whose energy decreases until exhaustion with the node activity. The second constructor defines a battery fulfilling the particular characteristics associated to COSA.

The conjunction of the *CfAbstractBattery* and *WaspMoteBattery* classes complete the definition of the power supply element of a COSA node.

6.2 Communication Modules

COSA definition relies on the agent's communication capability. Agents exchange information through local communication and arrange themselves in coalitions. Then, only one agent per coalition sends its sampling data to the server of the network.

To model this situation, we distinguish two communication modules for a COSA agent: one module dedicated to local communication and another one for communication with the sink. These communication interfaces do not interact with each other. The set of messages that can be sent are also defined. The composition of this module is presented in Figure 6.2.

6.2.1 Communication interfaces

Separate management of messages aimed at local communication from those addressed to the sink favours an easier development and analysis of the MAS application. As explained in Chapter 5, the communication module presented in Section 5.3.2 offers all the basic functionalities that a node may need. To define regular communication between neighbouring agents, we use the *Radial* communication module; whereas for the emission of messages addressed to the sink, we rely on the *Wireless* communication module.

The Radial communication module of a COSA agent is created from the general *Radial* module. All Radial elements, but the transmitter, are adopted as their general definition. However, the *RadialTransmitter* definition has to be adapted to the particular COSA's battery management; given that this element is responsible for emitting the last disconnection message. The *RadialTransmitter* provided by ECA-WSN module cannot work when the energy level of the battery is below the E_{sl}. To overcome this difficulty, we define the *CfRadialTransmitter* class.

CfRadialTransmitter extends from *RadialTransmitter*. This class manages the emission of the disconnection message addressed to neighbouring agents.

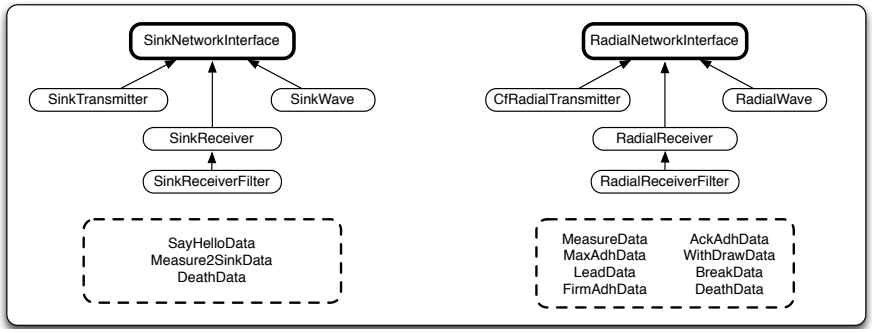


Figure 6.2: Communication module.

Then, these neighbouring agents are warned about the neighbour exhaustion and can act consequently.

Finally, and as its name states, this *Radial* communication module models omnidirectional communication that fades with distance. Therefore, its complete definition will require the specification of consumption, bandwidth and sensing radius parameters.

To model a COSA agent communication with the sink, we define the *Sink* communication module. All its elements extend from their equivalent *Wireless* elements (*SinkNetworkInterface*, *SinkTransmitter*, *SinkReceiver*, *SinkReceiver-Filter* and *SinkWave*). The only difference between the elements created and their corresponding *Wireless* ones refers to the phenomenon to which they are associated, that is, the *SinkWave*. Nonetheless, the *SinkTransmitter* presents more differences.

Changes presented in the *SinkTransmitter* aim at managing the consequences of battery depletion according to COSA. As explained previously, when the battery capacity reaches the E_{sl} level, the agent normal thread of action is interrupted. The only permitted action since that moment is the emission of the disconnection message, which is sent through the *RadialNetworkInterface*. Local and sink communication modules do not interact. Hence, to notify the sink about the node depletion, we have to artificially send a *DeathData* message through the *SinkTransmitter* without any cost.

As the *Sink* and the *Radial* communication modules model the same communication device, their technical characteristics are set to the same value. Nonetheless, for the sake of simplicity and to focus on COSA behaviour, we discard modelling the typical multihop message routing of WSNs that would correspond to the *Sink* module. Instead of this, the evaluation of a message sending is based on the square of the agent's distance to the sink. Analogously, the consumption associated to *SinkReceivers* is set to zero as agents are not supposed to receive messages sent to the sink.

6.2.2 Communication messages

A set of data communication messages is defined to simulate the desired application. Most of the defined messages appear to model the specific communication characteristics associated to COSA's negotiation protocol, but general communication messages are also included. In any case, the role associated to each message is easily identifiable from its name.

SayHelloData

SayHelloData class represents the message sent by sensing agents to the sink when they turn on. The aim of this message is to inform the sink about the fact of a node joining the network. This message contains information about the node position.

Measure2SinkData

Measure2SinkData class appears as a consequence of the two different communication interfaces in the node. This class represents the message that a leader agent sends to the sink to inform about the sample value collected. This message also contains the list of neighbours for which the agent works.

MeasureData

MeasureData class corresponds to the data message that an agent sends to inform its neighbours about how it perceives the environment. Typically, this is the broadcast message that initiates a negotiation to establish a leader-follower relationship. This kind of message carries information about the last sampled value from the environment and the variable's model it assumes, that is the mean and sigma parameters characterising the distribution. It also contains the timestamp associated to the sample sent. Besides this data, the *MeasureData* includes the current lead capacity of the agent. Thus, if the recipient of the message is a follower of this agent, it can take advantage of this information to update the information about the relationship that they hold.

MaxAdhData

MaxAdhData class represents the second message that would be sent in a generic dialogue. It represents the answer to an interesting *MeasureData* and contains the *maxAdherence* value. This message is also used by its sender to communicate its perception of the environment to the addressee neighbour. The addition of its perception can be easily done as this class extends from the *MeasureData* class.

LeadData

LeadData class, as its name states, contains the node's lead information. This class has two properties. One of the properties corresponds to the current leader-

ship value of the agent according to its established leader-follower relationships. The other property represents the agent's leadership value corresponding to the potential situation if this negotiation succeeds. Analogously to the *MeasureData*, including information about the current attitude of the agent allows for negotiations based on up-to-date data.

FirmAdhData, WithdrawData, BreakAdhData and DeathData

These *Data* classes are all exactly the same except that by their name, which associates them to a particular negotiation stage.

The *DeathData* message informs the agent's neighbours and the sink node about the sender node's immediate exhaustion, so that they can act consequently.

AckAdhData

The *AckAdhData* class, unlike the classes just presented above, includes a piece of information referring to the new leadership value that this node assumes after just confirming this relationship. Thus the recipient agent initiates its sleeping period after receiving the latest data about its relationship with its leader.

The set of classes described provides COSA agents with the communication capabilities that they require to actuate according to the algorithm.

6.3 COSA utils

An agent that implements COSA needs to be able to store and process the information it receives. The following classes and methods provide a COSA agent with the functionalities it needs to follow the algorithm.

6.3.1 Mathematical functions

The *MathematicsFNeighInfoTimeStamps* class contains, as it own name states, all the mathematical methods required by a COSA agent. These methods can be divided into two simple categories: COSA methods, which specifically represent COSA characteristic mathematical functions; and auxiliary methods, that support the previous ones. Figure 6.3 shows an outline of this class and its methods.

Auxiliary methods are *normpdfJava()*, *entropyNormalCalculation()*, *normalizedEntropy()*, *meanJava()* and *stdJava()*. The first method, *normpdfJava()*, allows for the evaluation of the Normal probability density function of an input value. In the same vein, the *entropyNormalCalculation()* returns the entropy value associated to a certain Normal distribution. As it was explained in Chapter 3, it is important to guarantee a minimum quality of the distributions involved in the adherence evaluation. The *normalizedEntropy()* method performs this task. This method just mimics the mathematical formulation proposed for this purpose. Finally, the last two auxiliary methods presented, *meanJava()* and

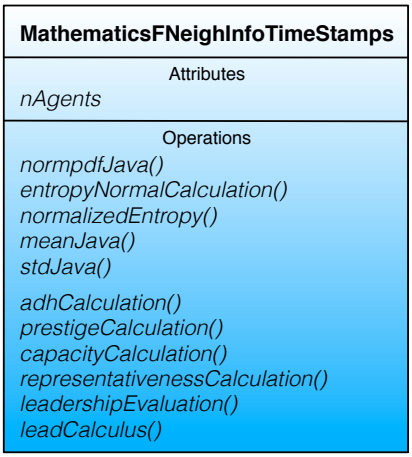


Figure 6.3: *MathematicsFNeighInfoTimeStamps* class outline.

stdJava() allows for the obtention of the mean and standard deviation associated to a set values, characterising these set of values as a Normal distribution.

As it is expected, COSA methods evaluate the adherence between two agents and each of the factors that define their leadership attitude. The set of methods included in this group are: *adhCalculation()*, *prestigeCalculation()*, *capacityCalculation()*, *representativenessCalculation()*, *leadershipEvaluation()* and *leadCalculus()*. These functions rely on the methods previously presented to perform their actions.

Grouping into a class all the mathematical methods that a COSA agent may need eases possible enhancements of the algorithm. Changes on this aspect of COSA would mainly influence this class, but not the inner structure of the node, what favours the reusability of the rest of components.

6.3.2 Information storage

In the same way that the math module supports the mathematical activity of a COSA agent, two other classes are introduced to help agents develop their function. These classes are *NeighInfoTimeStamps* and *RotatingQueue*.

The *NeighInfoTimeStamps* class represents the information store of the agent. This class contains numerous properties referred to the own node characteristics and its perception of the environment, including both, the observed variable and the agent relationship to its neighbours. The set of properties contained can be identified in Figure 6.4. This class also presents methods for accessing the information kept and updating the value of these properties whenever it is required.

The *RotatingQueue* class defines another necessary object for the agent to perform its action. This class maintains a LIFO structure that holds a certain number of the last samples collected by the agent. This structure represents the

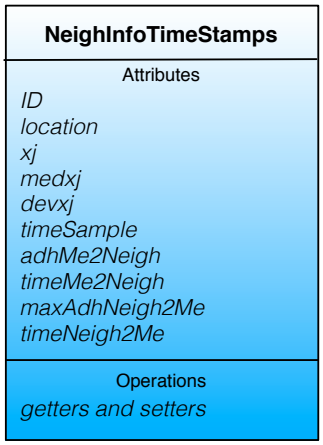


Figure 6.4: *NeighInfoTimeStamps* class outline.

agent's sampling activity memory. The size of this memory is specified by the programmer according to his interests. The utility of this class becomes clear when the agent updates its model of the observed variable by adding a new sample and discarding the oldest one.

6.4 Agents

The definition of an agent that behaves according to COSA concerns the final implementation of the core of the algorithm. Sensing agents following COSA are created from the conjunction of the following two classes: *AbstractCfAgent* and *SensorAgentSimple*. The *AbstractCfAgent* class aims at configuring the agent inner structure in charge of managing the reception of *SimulationEvents*. On the other hand, the *SensorAgentSimple* properly contains the COSA engine providing the intelligent behaviour. This approach favours the split between COSA intelligence and other agent's functioning tasks.

6.4.1 AbstractCfAgent

As its name states, the *AbstractCfAgent* class represents an intermediate class between the class that will properly hold COSA characteristics and the *AbstractSimulationAgent* provided by the platform. This non-instantiable class provides the agent with some features required by the application, but without implementing the algorithm yet. These characteristics refer to the particular kind of battery COSA agents need to use, to the temporal cost of agent's actions and also to the management of the different *SimulationEvents* arriving at the agent.

The *AbstractCfAgent* is associated to a *CfAbstractBattery* element. Therefore, an agent of this class has access to the particular features of this kind

of battery. Other properties included in this class represent the temporal cost associated to actions involving the CPU and inner communication between the node's components. Among the considered CPU actions, we can cite reading a sample or computing the adherence value, for instance. Figure 6.5 shows an outline of this class.

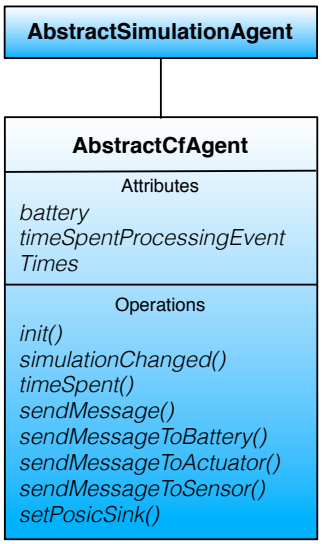


Figure 6.5: *AbstractCfAgent* class outline.

This class overwrites two methods inherited from *AbstractSimulationAgent* to include the properties just mentioned. The *init()* method in this class sets the values of the CPU consumptions and the *deathThreshold* of the *CfAbstractBattery* element.

The method in charge of managing the *SimulationEvents* is the *SimulationChanged()*. The new implementation of this method takes into account the CPU consumption properties and the presence of a *CfAbstractBattery*. The process of an event now demands an initial distinction between *DeathEvent* and any other kind of event. The reception of a *DeathEvent* requires the use of the *isCriticalEmpty()* and the *consumeCriticalEnergy()* methods. The rest of events check the emptiness condition regularly and consume energy normally. The evaluation of the energy spent in the event processing and the associated action performance relies on the *timeSpentProcessingEvent* property and the *timeSpent()* method, which returns the time invested in this whole process. The methods that adopt the properties referring to the agent inner communication temporal costs are *sendMessage()* or *sendMessageToActuator()* among others. Hence, this time is also taken into account when processing their corresponding events.

Another simple but characteristic method required for experimentation is the *setPosicSink()*. This method codifies the sink position in the agent. This artefact is introduced to allow agents to evaluate their distance to the sink and

then know the energy consumption associated to this action.

6.5 SensorAgentSimple

This is one of the most complex classes of the application, as it contains the reasoning core of COSA. The *SensorAgentSimple* represents the CPU of the node. Hence, this class knows all the other elements composing the node: communication module (*Radial* and *Sink* elements), battery (of *WaspMoteBattery* kind) and the sensor used to monitor the environment, that is, the *RiverPollutantPhenomenonSensor*.

In order to handle the information that a COSA agent needs, this class contains different types of properties able to save received or collected information. These properties can be grouped into different sets regarding its goal. There are properties aimed at keeping the model of the environment, and the information collected, properties about the neighbours' information, properties related to the role played by the agent in the negotiation process and there are also properties for each of the parameters defined in COSA. Among these parameters are, for instance, those related to the admissible extreme values for standard deviation. Some of these properties, together with the methods contained in the class, can be observed in Figure 6.6. To describe this element, and how COSA is implemented in it, we present methods aimed at initialising the element, methods for processing *SimulationEvents* arriving at the agent and methods for sending and processing received messages separately.

6.5.1 Setup and initialisation

The *setup()* and *init()* methods build and initialise the node by creating its different parts: the *WaspMoteBattery*, the *RiverPollutantPhenomenonSensor* and the *RadialNetworkInterface* and *SinkNetworkInterface*. The creation of elements of these types implies the specification of their associated properties, such as *deathThreshold* or *samplingFrequency*. The communication module is a little more special as it is composed of two interfaces. The *bandwidth* property of both transmitters is set to the same value (as they represent the same physical element), but the cost per action and unit time is different for each transmitter, as explained in Section 6.2. We assume that local communication happens only between neighbour agents situated one-hop distance away, whereas communication with the sink takes into account the distance (in radio hops) to this element. The standby consumption of the communication module is modelled through the *RadialReceiver* element. The *SinkReceiver* is not even activated as COSA agents do not hear messages addressed to the sink.

Regarding the agent configuration for the use of COSA, the *init()* method fixes the initial conditions of the agent. A COSA agent initiating its performance is alive, but it is asleep (not capable of sampling or negotiating). The agent considers itself as its own leader and assumes null or non-admissible values for the variables involved in between neighbours relation. Behaving according to COSA

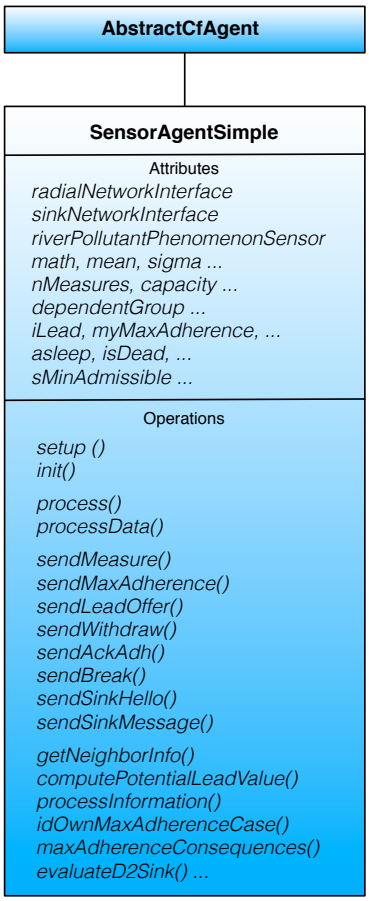


Figure 6.6: *SensorAgentSimple* class outline.

requires the agent to have a model of the phenomenon being observed. This Normal model of the environment is also preset here, and a random distribution is used to initialise the *RotatingQueue* memory element of the agent.

Finally, the *init()* method calls the *SendSinkHello()* function to inform the sink about the presence of this agent in the network. The emission of a *WakeEvent* to change the asleep state of the agent within the next five minutes is also programmed here.

Every element composing the node (sensors, actuators and battery) share a unique identifier with the agent. This identifier results very useful to debug the application and trace the behaviour of the agent.

6.5.2 Events processing

Communication between the node's components takes place through the exchange of different kinds of *SimulationEvents*. A *SensorAgentSimple* agent can receive three kinds of events, which are: *WakeEvent*, *MeasureEvent* and *DeathEvent*. The *SensorAgentSimple* class implements *process()* methods for each of them.

The *WakeEvent* event is sent when the agent is being initialised and at the end of a sleeping period. The processing of this event consists of changing (if necessary) the state of the agent from asleep to awake, and of switching on the phenomenon sensor.

The phenomenon sensor sampling action, and the following reception of a *MeasureEvent* by the agent corresponds to the execution of the *Sense and Send* thread described in Algorithm 2. The reception of a new measurement causes the emission of a *PropertyChangedEvent* and the subsequent update of the agent information regarding the last sample obtained and the time of this action. The model of the environment (mean and sigma) is also updated according to the agent's samples memory. These actions entail a temporal cost that is taken into account when evaluating the energy cost of this process.

Once the agent's registers have been updated, the collected measurement has to be sent to the agent's neighbours. The *sendMeasure()* method performs this action. In case the agent plays a leader role, it also sends this information to the sink through the *sendSinkMessage()* method. These sending methods will be explained later.

The reception of a *DeathEvent* coming from the *WaspMoteBattery* implies changing the state of the agent to dead. The process of this event also implies creating new *DeathEvents*. These events are used to inform the transmitters of the node that they have to send the agent's disconnection message, the *DeathData*.

6.5.3 Message sending

The execution of COSA by an agent makes it maintain dialogues with its neighbours to exchange information. The following methods allow the agent for the emission of different kinds of messages.

sendMeasure()

This method aims at creating and sending a broadcast message to all those neighbour agents situated within sensing radius distance. Every time that the agent takes a sample from the environment, it creates a *MeasureData* message. Besides the sample, the *Data* message also contains information regarding the model of the environment assumed by the agent, its leadership value and the time of transmission.

sendMaxAdherence()

A *MaxAdhData* message is emitted at the beginning of a negotiation process between two agents. As already explained, it adds the value of adherence to all the information contained in a *MeasureData* message. This method creates the *Data* message and sends it to the interested addressee.

sendLeadOffer()

Following the scheme of previous sending methods, the *sendLeadOffer()* creates and sends to the corresponding neighbouring agent information about how good the agent would be if it were its leader. This message also includes information about the current leadership attitude of the agent. The addition of this item avoids deadlock situations caused by an agent negotiating with its current leader whose position may be worsening.

sendWithdraw()

This method creates and sends a *WithdrawData* message to the agent's leader. No information justifying this action is included in the message.

sendFirmAdherence()

The *FirmAdhData* that this method creates is an empty message that confirms to the addressee neighbour the agent intention of being led by it.

sendAckAdh()

This method builds the corresponding *AckAdherence* message and sends it to the corresponding agent. This message contains the leadership attitude of the agent at the right moment of confirming the leader-follower relationship.

sendBreak()

As a follower, an agent can break its relationship with its leader through a *WithdrawData*, so can a leader with a *BreakAdhData*. This method creates and sends this message.

All the methods described send messages through the *RadialNetworkInterface*. That is, the messages created are all intended at inter-agent communication. Nonetheless, agents also communicate with the sink to send the information collected through the *Measure2SinkData*. As mentioned in Section 6.2.2, a *SayHelloData* is also sent to the sink to inform it about the agent joining the network and its position. These messages are sent through the *SinkNetworkInterface*. Methods building these messages and sending them are ***sendSinkHello()*** and ***sendSinkMessage()***.

6.5.4 Message reception

When an agent behaving according to COSA is not asleep, it exchanges information with its neighbours to find out its preferred situation in the organisation. The reception of different messages from neighbours triggers different sets of actions by the agent. Methods aimed at processing data messages share the *processData()* signature, but they differ in the kind of messages they can admit. The actions that each of these methods performs conform to COSA *Information Processing* thread presented in Algorithm 3 on page 36.

The *processData()* associated to a *MeasureData* message is executed if and only if the agent is awake. The actions derived from the reception of this kind of message are handled by two auxiliary methods: *processInformation()* and *maxAdherenceConsequences()*. The first method collects the information received in the message and then identifies the possible interest of the agent in the neighbour that sent the message through the *idOwnMaxAdherenceCase()* method. The *maxAdherenceConsequences()*, as its name suggests, triggers the consequent actions derived from the previous method.

Processing a *MaxAdherence* message requires the agent to be awake too. This method includes the actions corresponding to the reception of a *MeasureData* plus those specific to this kind of message. These specific actions are the agent self evaluation as a leader of the node which sent the message and, then informing it about this evaluation. Methods used for this purpose are *computePotentialLeadValue()* and *sendLeadOffer()*.

The *processData()* method associated to a *LeadData* makes the agent check the precedence of the message. If this message comes from its current leader, the agent updates its current state. Then, the agent checks if the offer improves its situation and acts consequently by sending, or not, a *FirmAdherence* message.

The reception of a *FirmAdherence* message confirms the recipient about the intentionality of the neighbour of becoming part of a coalition led by it. This situation entails reevaluating the lead attitude of the agent. If the agent still prefers becoming a leader of the neighbour, and it is a follower, it breaks this relationship and updates the relationship to its neighbours. Finally, an *AckAdherence* message is sent to the corresponding neighbour becoming now its follower.

An *AckAdherence* message aims at changing the state of the recipient. Hence, if the agent receiving this message was a leader, it stops being a leader and dismantles its group to become a follower on this message's sender. In the case that it was already a follower of a different agent, it changes its leader agent. As a consequence of being a follower node, it switches off its phenomenon sensor and plans a *WakeEvent* after the sleeping period. These actions take place if and only if the message does not arrive out-of-step that is, the initial conditions of the negotiation still hold.

An agent who plays a leader role can lose its followers when it receives a *WithdrawData* message. The processing of this kind of message implies the deletion of the sender entry from the set of followers of the leader agent (dependent group) and the corresponding update of the leader state variables.

Follower agents can receive *BreakAdhData* coming from their leaders. This kind of messages is always heard and processed by their addressees regardless of whether they are asleep or awake. Its processing takes the agent to a new leader state by breaking its link to its previous leader (message sender) and switching on its sensor.

Finally, the process of a *DeathData* implies the deletion of the information associated to the dying agent and the removal of the relationship with it. As it happens with *BreakAdhDatas*, *DeathDatas* are processed by the agent independently of the node asleep or awake state. As a consequence of the reception of a *DeathData*, and depending on the kind of relationship established between the addressee agent and the sender, a set of actions is performed to allow the recipient agent to continue its performance.

Message processing relies on different auxiliary methods. These auxiliary methods extract information from the messages received, perform mathematical functions, compute the temporal cost of the actions or evaluate the agent state after the message reception. For instance, we can cite *computePotentialLeadValue*, among others appearing in Figure 6.6. The class shown in this figure can develop an intelligent behaviour that satisfy COSA principles.

6.6 COSA strategies

The *SensorAgentSimple* class described in the previous section follows the formal definition of COSA as presented in Chapter 3. This basic definition is open to some modifications that may help in adapting it to different environments or tasks. In this section, we propose two strategies that are used by COSA (in isolation or combined) and that lead to different behaviours of the COSA agents. Changes introduced in the strategy influence the balance between the energy consumption and the overall observed error of the WSN. However, the following strategies do not represent an essential change of the algorithm, they just slightly alter the agent behaviour when certain circumstances are met.

6.6.1 Sampling Frequency

The *Sampling Frequency* strategy alters as its name states, the fixed sampling frequency of leader agents. COSA presentation and implementation assumes that the sampling frequency of the environment is a parameter fixed by the programmer. This strategy removes this assumption and provides the agents with a very simple individual adaptive sampling strategy. According to this strategy, when a leader agent has three or more follower agents, it doubles its sampling frequency. The aim of this strategy is to make COSA agents grouped in a coalition to be more reactive to changes on the environmental conditions. The implementation of this strategy leads to an earlier perception of changes, therefore, to smaller deviation times.

The implementation of this strategy barely affects the *SensorAgentSimple* class. This new feature is introduced into the agent through the *processData()*

method associated to *FirmAdherence* messages. When an agent receives a *FirmAdherence* message, it counts the number of neighbouring agents that form its dependent group. If the size of the group is over two, the agent automatically doubles its sampling frequency. Although the initial definition of this strategy is fixed, the number of *agents per group* (*apg*), triggering the frequency change and the *multiplication factor* (*mf*), of the frequency represent COSA *Sampling Frequency*'s configuration parameters. Thus, when this COSA strategy is implemented, these new two parameters have to be included in the p parameters set defined in Section 3.1.

6.6.2 Coherence

This strategy checks whether the leadership condition of an agent is still coherent with the last sampled values it had. The behaviour of a leader consists of sampling the environment, updating its model of the environment and sending the sampled value to the sink as the value representing all the members of the coalition. However, if the updated model differs from the model that the leader had when a member of the coalition joined in it is unclear whether the agent (in sleeping mode to save energy) would still be willing to stay within the coalition. Thus, this strategy allows a leader agent to evaluate the difference between the just sampled value and the previous sample. If this difference is significant enough, over a threshold V_T, the leader agent proactively wakes its followers so that they can sample the environment and decide again which coalition to join. As it happened with the COSA *Sampling Frequency*'s parameters, V_T is a configuration parameter associated to this new strategy. Therefore, V_T has to be added to the initial p parameters set identified in Chapter 3 when considering COSA *Coherence* application.

The goal of this strategy is again to be more reactive to changes in the environment. As soon as there is a drift in the sensed values of a leader the follower agents will be waken up to sample again as the coalition raison d'être (similarity of sampled values) may be at stake.

The inclusion of this strategy in a COSA agent behaviour requires little changes on the *process()* method of a *MeasureEvent*. The right moment to test the *coherence* condition is when the leader agent performs a sampling action. Together with the typical process associated to the sample, the difference with the last sampled value can be evaluated. As the strategy indicates, important differences between these values lead to the emission of *BreakData* messages to follower agents. Except for this change, the COSA agent thread of action conforms to the basic definition.

Both strategies increase the sensing and thus the energy consumption with respect to the basic COSA operation. The computational effort implied by these strategies is negligible although certainly the number of messages exchanged and the number of sampling actions taken by leaders and coalition members grow. The adequacy of these techniques depends on the kind of environment to be monitored and the interests of the programmer or network manager.

6.7 SinkAgent

This *SinkAgent* represents the central node that receives the samples collected by the monitoring agents composing the network. This element holds the tools to process the information gathered. Therefore, its implementation is strongly conditioned by the information evaluation criteria of the programmer or application manager. Despite this fact, we include the definition of the *SinkAgent* class in the COSA-able WSN module as without the server; the network would be incomplete, and its definition is better explained in this context. The criteria for information evaluation implemented in this class are specifically detailed in Chapter 7. They mainly affect a particular method of the agent class. Therefore, changing the evaluation criteria entails replacing this method by the one adapted to the new specifications.

The class structure repeats the *SensorAgentSimple*'s scheme as shown in Figure 6.7. It contains a set of general properties to keep the information received and its corresponding evaluation. Moreover, it also contains different parameters associated to the evaluation formulas implemented.

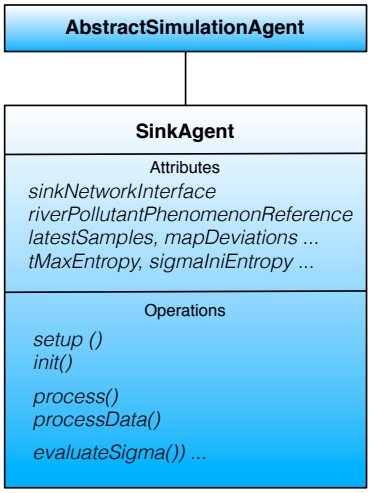


Figure 6.7: *SinkAgent* class outline.

6.7.1 Setup and initialisation

The *setup()* and *init()* methods create and initialise all these properties and node components. The *SinkAgent* runs on an *InfiniteBattery* element as we assume that it is connected to the net. This agent class just perceives the information sent to it and processes it. Hence, its communication module can be defined from a unique *SinkNetworkInterface*. The consumption associated to the communication module and the CPU is set to zero as it does not have any effect on a battery of infinite capacity.

The implemented evaluation of the information received bases on the real value of the monitored phenomenon in the environment. Therefore, we include in the *init()* method the definition of a direct access to the phenomenon observed, as an artificial tool for application purposes. This artefact allows the *SinkAgent* to have its own view of the phenomenon and to evaluate the distance between the information collected by the sensors in the network and reality at any location.

Moreover, the *init()* method triggers the sending of *UpdateErrorEvents*, whose *process()* method performs the periodic evaluation of the data received at the sink.

6.7.2 Events processing

The only *process()* method included in this class is the method associated to the *UpdateErrorEvent*. This method is the most complex one of the class and the one on which to act to change evaluation criteria.

The network performance is evaluated in terms of committed error, and entropy associated to the nodes composing the network. Each time an *UpdateErrorEvent* is received, the value of these magnitudes is calculated. The error evaluation relies on fulfilling a map containing, for each agent in the network, the value perceived and the actual value of the phenomenon at that point. With this information, the calculation of the error committed by each agent and the global mean value is quite easy.

The quality of the information available in the network is calculated as the addition of agents' entropy measures. This evaluation relies on the entropy values of the observed variable models of each agent and the time elapsed since the agents last sample collection. Both evaluations use information saved to properties included in the class.

6.7.3 Message reception

The *SinkAgent* receives information to evaluate through *Data* messages that it can process. The kind of messages that can be received by the *SinkAgent* are *SayHelloData*, *Measure2SinkData* and *DeathData*.

SayHelloData is the first message received by the *SinkAgent* from any of the nodes in the network. The process of this message is used to obtain information about the node identity and its location. This message helps the *SinkAgent* composing its view of the network deployment.

Processing *Measure2SinkData* messages renders information about the nodes' perception of the observed phenomenon. When one of this kind of messages is received, the sample collected and the time of reception are saved to the *latestSamples* and *latestSamplesTimes* properties. These values are saved related to the sending agent and, if that is the case, those other agents depending on the sender, i.e., the rest of coalition members.

The last kind of message receivable by the *SinkAgent* is *DeathData*. Receiving a disconnection message from an agent implies deleting its entry from

the *SinkAgent*'s map of the network, as this node will no longer provide any information.

The *SinkAgent* definition completes the network function, as it presents the last actions performed on the information collected by the sampling agents.

6.8 CfAbstractReport

The *CfAbstractReport* class defines basic characteristic for information extraction from COSA simulations. Capturing information from a simulation requires the application definition completion to identify the interesting features to monitor. Nonetheless, knowing the characteristics of the behavioural strategies that agents can implement, we introduce the *CfAbstractReport*.

The *CfAbstractReport* extends from the *SimulationReport* provided by the RepastSNS platform and defines basic characteristics common to every kind of report. This class contains general methods to manage the files to generate. It links the report to the particular *SimulationModel* being simulated and also establishes the name pattern for the generated files. Particularly, files are named after the simulated model, the observed variable and the specific simulation seed. Hence, once the interesting simulation features to monitor are selected, the definition of their associated report focus on capturing and processing the corresponding *BasicAction* and implementing the *actionPerformed()* method.

6.9 Conclusions

The COSA-able WSN provides with the COSA intelligence required for the application simulations. This module identifies how COSA implementation affects the different elements composing a node and also the simulation structure itself. The definition of COSA-able WSN module follows RepastSNS principles with respect to extensibility and scalability. Hence, the implementation of COSA and its different facets relies on independent yet related classes structures. The importance of this design stands out when the proposed COSA alternative strategies are presented, as their introduction barely affects the previous work developed. Ultimately, to test COSA agents' behaviour it is still necessary to implement another layer that holds the environment and establishes the simulation conditions.

Chapter 7

Experimentation

COSA aims at faithfully monitoring the state of a dynamic environment and at extending the lifetime of the network as much as possible. To test this, the scenario considered is that of a river, whose state is to be monitored. In previous chapters, the base structure over which to build experiments has been presented. To define the desired experimental setup, two additional modules have to be built: one, to model the environment where the network is deployed, and another one adapting a COSA-able WSN to the domain. The completion of these tasks leads to the next step of evaluating the algorithm proposed. The results derived from experimentation deliver information about the algorithm characterisation and performance, and about the methodology developed to reach them.

7.1 Riversim

Defining features of COSA and how it models an agent's behaviour have already been introduced in Chapter 3. Proper implementation of the algorithm into a simulated agent can also be found in Chapter 6. Concepts inspiring COSA definition refer to an intelligent use of the node resources when it develops its task of monitoring the state of a particular environment. Therefore, to evaluate COSA performance and its influence on the node's behaviour, we need to define the environment that the nodes survey and where they are deployed. Figure 7.1 recalls the development structure required for the creation of the application.

The Riversim module depicted in Figure 7.1 represents the application domain. Its definition is based on a set of classes that model the environment of interest.

As already mentioned, a river is the application scenario selected to test COSA performance. Different pollution sources appear in the monitored river. The aim of the network deployed on the river is to detect and inform the sink about these stains dynamics. In the following sections, we present the class structure defined to model the application domain correctly and to introduce the phenomenon to be monitored in the simulation platform.

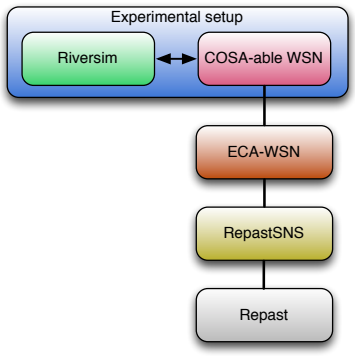


Figure 7.1: Development Structure.

7.1.1 Phenomenon

The definition of the phenomenon to be monitored represents the main task of the Riversim module. As it name states, the monitored environment is a river in which different pollution sources may appear along time. The composition of this phenomenon relies on two classes and their interactions. These classes are *RiverPollutantPhenomenon* and *Stain*.

Stains definition

To define a pollution stain appearing in a river, we consider a set of parameters that characterise all the features that a stain might have. When describing a pollutant, it is important to identify its location, its size and its intensity at the moment of appearance. A stain may last in time when it appears due to a contaminant source spewing for a period. In this case, the duration, the spewing pace and the flow of the stain are also considered.

The simulation class created, *Stain*, contains a variable for each of the properties enumerated, besides their corresponding access methods. Its structure can be observed in Figure 7.2(a). The introduction of pollution sources in the river takes place through an events' generation and processing schedule that will be described later. However, this generic process can be easily adapted to any other kind of phenomenon that may appear in a river.

In fact, we also define a different kind of pollutant source, the *StainSin*, that favours the creation of a more dynamic environment. Figure 7.2(b) shows the outline associated to this class. The *StainSin* allows for the introduction of sinusoidal pollutants in the river. These sinusoidal stains cause the presence of a continuously oscillating contaminant intensity where the stains are located and their surroundings.

The definition of the class associated to this kind of phenomenon follows the same principles as regular *Stain*. However, a *StainSin* object creates a sinusoidal stain. As a consequence, the dumped flow is not a constant value, but it is

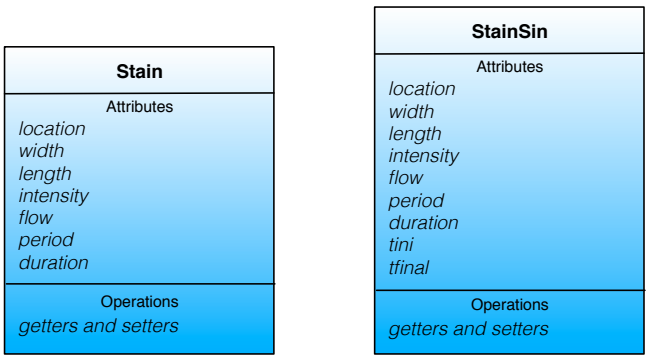


Figure 7.2: *Stain* and *StainSin* classes outline.

evaluated at each time instant according to the maximum intensity reachable for the stain and the period of the phenomenon modelled.

The introduction of this new class does not affect the structure the Riversim module, although it requires the corresponding adaptation to this particular phenomenon.

River Phenomenon

The river phenomenon represents the environment in which both, the stains to detect, and the nodes composing the network, coexist. Simulating a river environment requires the reproduction of a water flow. This river phenomenon is also in charge of managing the pollutant stains' appearance. Therefore, these two parts can be clearly identified in the corresponding simulation class modelling the element, the *RiverPollutantPhenomenon*.

- River movement imitation

The river phenomenon models a section of a river. In order to mimic the effects of water flowing through the river, we define a simple river movement schedule that causes that any phenomenon appearing in the river is shifted by the current of the water. The implementation of this model relies on the definition of a grid covering the whole river section and a set of variables specifying the stream's behaviour.

The model used to define the river movement considers two components: a drift component and a sedimentation component. Equation 7.1 represents the mathematical formulation of this model, distinguishing these two components. According to this model, part of the phenomenon remains at its origin due to the sedimentation component, whereas the rest flows according to the strength of the current. Formula 7.1 gives the value of the intensity of the phenomenon at a cell of the grid as a composition of the phenomenon value at this point and its upper-neighbouring cells in the previous time instant. Values assigned

to parameters ρ, α, β, and γ determine the kind of river modelled. The $(1-\rho)$ term sets the sedimentation component whereas α, β, and γ model the drift. Particularly, each of these three factors models the contribution of each of the upper cells that we consider somehow pour downstream. As the formula shows, we restrict the contribution of the drift component to a downstream cell to the three adjacent upper cells. Therefore, if any contaminant is poured in a water cell, it will spread to its downward cells through time according to the following equation:

$$River^t(x,y) = (1-\rho)River^{t-1}(x,y) + \rho(\alpha(River^{t-1}(x-1,y-1)) + \\ + \beta(River^{t-1}(x,y-1)) + \gamma(River^{t-1}(x+1,y-1))) \quad (7.1)$$

Differences in the spewing pace, duration and number of contamination sources appearing along the river, together with this characteristic river current movement is what determines the dynamics of the environment under observation.

This approach favours an easy implementation of the river movement into the *RiverPollutantPhenomenon* class. Figure 7.3 shows a basic outline of the class developed.

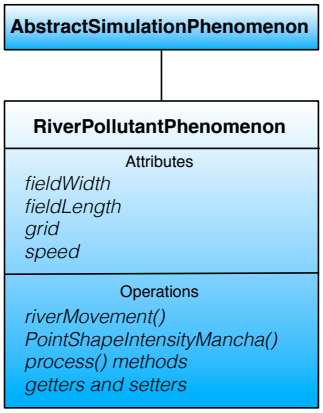


Figure 7.3: *RiverPollutantPhenomenon* class outline.

The constructor of the class creates the grid from the field dimensions and the size of the desired matrix representing the environment. *setup()* and *init()* methods start the river and its movement according to the parameters given. The simulation of the dynamic river flow is based on the periodic emission and processing of an especial *SimulationEvent*, which is the *UpdateEvent*.

An *UpdateEvent* is an empty class whose strength lies in its processing action. Every time that the *RiverPollutantPhenomenon* receives and processes this kind of event, the *riverMovement()* method is executed. As its name indicates, this method applies the formula Equation 7.1 over the grid of cells composing the

river and updates their values. This artefact makes this river component imitate the river flow.

- Pollution sources appearance in the river

The introduction of pollution stains in a clean river by the method described above is based on the consecutive emission and processing of particular events. These events deal with the nature of the stains and condition their appearance in the river according to the stain definition.

The events' structure supporting this is composed of four different classes *PollutionEvent*, *StainGenerationEvent*, *PeriodicStainEvent* and *KeepGoingEvent*. The reception and processing of these events by the *RiverPollutantPhenomenon* either create a pollution source in the river or update an existing one. Figure 7.4 shows how the events relate to each other and how the process of one of them implies the creation of the other.

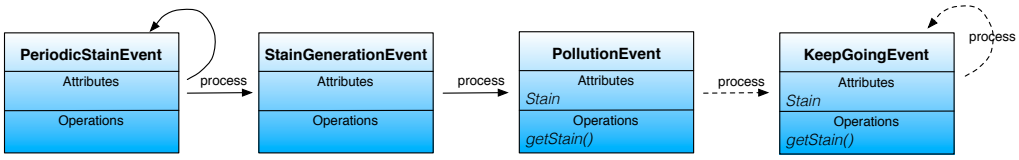


Figure 7.4: Events structure associated to pollution appearance.

PollutionEvent can be considered as the basic element of this structure. The definition of this class demands the specification of the particular *Stain* object to introduce. The process method associated to this kind of event properly sets the stain in the grid modelling the river. The *RiverPollutantPhenomenon*'s method providing this functionality is the *PointShapeIntensityMancha()*, that checks the *Stain* object characteristics and changes the river grid according to them. Once the stain is in the river, its duration is also checked. In case the stain lasts in time, the emission of a *KeepGoingEvent* associated to the stain is programmed according to the spewing pace. Processing a *KeepGoingEvent* reproduces the described tasks associated to the *PollutionEvent* but subtracts a cycle to the stain's duration. When the duration of the stain reaches zero, no more *KeepGoingEvents* are programmed. These two event classes, and their process methods, allow for the introduction of stains (instantaneous or with some duration) in the river.

The kind of stain to appear, regarding its sporadic or periodical character, is predefined together with the *RiverPollutantPhenomenon* specification. The event triggering the process of contaminating the river is designated when initiating the element, that is, in the *init()* method of the *RiverPollutantPhenomenon*. Hence, if the programmer wants the appearance of a one-shot stain in the river, a *StainGenerationEvent* has to be defined. Processing this event implies creating a stain and a PollutionEvent, whose processing will trigger the above described processes. On the other hand, to define a periodic stain, the *PeriodicStainEvent*

is used. In this case, its process creates a *StainGenerationEvent*, taking then advantage of the structure created. Furthermore, and as the stain to create will appear periodically, the emission of another *StainGenerationEvent* is programmed according to the stains' appearance periodicity. This value is hard codified together with the desired type of stain. Recalling the events' relationship shown in Figure 7.4, it is important to highlight that not the whole set of classes appears always, as it depends on the stain characteristics.

Hence, this interwoven but simple structure allows for the introduction of pollutant sources in the river. It models the interrelationship that links both parts of the river definition, the water flow movement and the appearance of stains. Differences in the spewing pace, duration and number of contamination sources appearing along the river, together with this characteristic river current movement is what determines the dynamics of the environment under observation.

As already stated in Section 7.1.1, the introduction of a different kind of pollutant sources can be considered. This election does not affect the events' structure, but it does require the creation of events adapted to the new kind of stain. As the sort of stains and their characteristics depend on the programmer's preferences, and these are introduced together with the phenomenon, the easiest option is to define a new phenomenon class including these changes. Thereafter, we define the *RiverPollutantPhenomenonSin* that mimics the river movement of the *RiverPollutantPhenomenon* and adapts the introduction of pollutant part to the *StainSin* phenomenon.

The definition of the *RiverPollutantPhenomenon* concludes the Riversim module. The elements described represent a river environment where a wireless sensor network implementing COSA can be deployed and with which it can interact.

7.2 COSA-able WSN adoption

The implementation of the Riversim module sets the application environment. The COSA-able WSN module defined a generic WSN able to behave according to COSA, that is, nodes composing the network behave in an intelligent way following COSA principles. Nonetheless, the link between these two modules needs to be established. Nodes sample the environment defined in Riversim, hence they need to be placed in this environment and their generic sensors have to adapt to it to be able to sample. As a consequence, the definition of particular sensors capable of collecting information from the specified domain constitute the main bond between both modules.

7.2.1 Nodes deployment

Placing the nodes in the field requires the definition of classes able to perform this task. The selection of a particular network deployment may be associated to the nature of the environment, the agent's sampling strategy or both.

Different classes have been defined to represent different layouts. All these classes extend from the *PhysicalDistribution* class defined in RepastSNS and they all follow the same structure too. *PhysicalDistribution* classes contain a set of properties related to field dimensions and/or specific distribution parameters, together with a *distribute()* method. Some of the classes defined are *UniformDistribution*, *ZigZagDistribution* and *MeshDistribution3Nodes*. As their names state, these classes place the nodes in the environment according to a particular pattern. Some of them place nodes at specific locations hard codified, while others distribute the nodes in the field according to its dimensions. In any case, these classes are associated to the generic *SimulationAgent* class, not being limited then to any particular kind of agent.

7.2.2 Sensors

As already mentioned, monitoring the phenomena described in the previous section requires the definition of proper sensors. In this case, the kind of sensor considered takes samples from the environment periodically and at specific times to check the presence of hydrocarbon pollutants in the river. Therefore, the defined sensor is a discrete sensor that adjusts to the generic outline introduced in Chapter 5. Its particularisation demands the definition of a filter specific to the phenomenon being monitored besides the setting of consumption and action specifications according to the device modelled.

Regarding the class implementation for simulation and assuming that the phenomenon to be monitored is the *RiverPollutantPhenomenon*, we define the *RiverPollutantPhenomenonSensor*. The sensor action specifications are added to the *setup()* method of the *RiverPollutantPhenomenonSensor* class, as well as the use of the *RiverPollutantPhenomenonSensorFilter*.

This class follows the standard features defined for a basic sensor, however it introduces a novelty compared to the standard definition provided by the *AbstractDiscreteSensor* class. In order not to create an ideal sensor, it includes a *noise* property which slightly alters the measurement actions of the sensor. This *noise* property relies on a Normal distribution function which is defined from the technical specifications of the sensor inspiring this class.

The effect of this modification becomes relevant when the *sense()* method is executed. This method returns now the collected value of the phenomenon at the sampling point plus a disruption given by this *noise* variable. The class created to implement this *noise* will be presented next.

Finally, the specific filter definition just requires designating the *RiverPollutantPhenomenon* as the phenomenon of interest, so that the so-called *RiverPollutantPhenomenonSensorFilter* can accept no other kind of event.

Once again, if we change the phenomenon of interest, we need to redefine the two classes presented to adapt them to the phenomenon. *RiverPollutantPhenomenonSinSensor* and *RiverPollutantPhenomenonSinSensorFilter* give an example of how this can be done.

7.2.3 Normal Distribution

The description of a class that characterises a normal function relies on a very simple interface and a class definition.

The *Distribution* interface provides with a general framework to model different distribution functions that can represent different phenomena. This interface just declares a *nextValue* method to return the corresponding value of the distribution function.

The *NormalDistribution* class implements the *Distribution* interface. As it can be observed in Figure 7.5, *NormalDistribution* contains three properties that characterise the distribution. The mean and deviation values specify defining parameters of the Normal distribution, and a Random object that is in charge of generating the phenomenon values. Properties mean and deviation are specified when the object is created through its constructor.

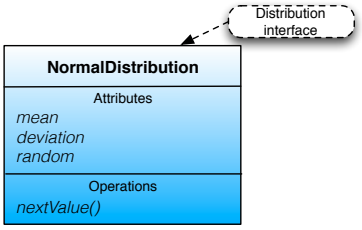


Figure 7.5: *NormalDistribution* class outline.

7.3 Simulation tools

The goal of the simulation is to reproduce a set of conditions in a controlled environment to study the performance of the elements involved and to get information for further development. To eventually set up the target simulation of a WSN implementing COSA deployed in a river scenario, there are still two more tasks to perform. The first task consists in identifying the information to obtain from the simulation and then, to define associated reports. The second task requires arranging a *model* class that binds all the elements taking part in the simulation (agents, phenomenon, report, etcetera). Once these tasks have been completed, simulations can be executed and results stored for analysis.

7.3.1 Simulation reports

Reports represent the tool that the programmer has to get information from the simulations. One distinctive characteristic of the simulation platform developed is the use of *SimulationEvents* for inter elements communication. A basic definition of reports allows to capture these *SimulationEvents* in order to extract the desired pieces of information.

The programmer may be interested in monitoring the behaviour of a particular simulation element or the evolution of a system's variable. In order to do this, the first task to accomplish is the identification of the *SimulationEvent* that contains the desired information. Then, it is necessary to define an associated report class able to capture this *SimulationEvent*. In case that the desired data is not directly accessible through an existing *SimulationEvent*, the programmer can introduce one *SimulationEvent* artificially for monitoring purposes.

All reports behave in the same way: they capture *SimulationEvents*, check their kind and emitter and, if all conditions are met, the desired information is extracted and printed to a log file. However, depending on the kind of event, the associated information and the emission frequency, its management can be more or less complex.

As it will be explained later in this chapter, to evaluate the performance of COSA in the scenario of interest, we rely on the quality of the information reported at the sink and the energy available in the network. Besides this, we are also interested in getting a general view of the network deployment when the simulation starts and in knowing the death pattern of the nodes. Consequently, the report classes developed focus on these aspects of the simulation. They all follow the pattern presented in Section 6.8 that particularise the general report definition provided by RepastSNS to COSA characteristics.

7.3.2 Report classes

To obtain a general perspective of the network after its deployment, an additional *SimulationEvent* is introduced. The *NetworkInfoEvent* is not a proper application event, as its purpose is to attach deployment data so that it can be monitored. This event carries information about the nodes' identifiers and position. The associated report class *NetworkInfoReport* just captures the corresponding event and generates the associated log file.

The *DeathEventsReport* class is very similar to the *NetworkInfoReport*. In this case, it is associated to a proper application event, such as the *DeathEvent* emitted by every node before depleting its battery. This kind of events appear at particular moments of the simulation and the report just registers its occurrence.

EntropyReport and *ErrorReport* classes monitor the evolution of the application in terms of these properties, entropy and error. Whereas the *EntropyReport* captures events specifically created for it (*EntropyReportEvents*), the *ErrorReport* uses a particularisation of a general RepastSNS event, a *PropertyChangedEvent*. Particularly, the *PropertyChangedEvent* referred to the error variable of the sink. This fact demands a refinement of the identification process of the event of interest by checking the sort of property contained in the event.

Both *EntropyReportEvents* and *PropertyChangedEvents* associated to the error property are emitted periodically. Each time they are generated, their value is registered in their associated log file.

Finally, *RemainingEnergyPerBatteryReport* and *RemainingEnergyReport* are aimed at monitoring the evolution of the energy available for each node and the

whole network. Tracking these values required the capture of the *PropertyChangedEvents* emitted every time that the batteries change their value. Unlike the events associated to the error, changes on the energy value of the nodes' batteries do not happen periodically what originates an enormous amount of information. These report classes preprocess the data to deliver mean or actual values of the network (*RemainingEnergyReport*) or individual batteries' energy periodically (*RemainingEnergyPerBatteryReport*).

Besides the already presented report classes, more classes have been created for application information retrieval and debugging purposes and can be enabled when necessary.

7.4 Simulation Arrangement

As explained in Chapter 4, the *SimModelImplSNS* provides a generic implementation of the element capable of binding all simulation elements together. *TheModel* class, which extends from *SimModelImplSNS*, is in charge of setting up and starting the simulation defined by COSA implementation in a river scenario. This constitutes the last implementation work to do to finally complete and tune the application to test. Figure 7.6 shows a simple outline of the class created indicating the methods that have to be overwritten to bring these elements together.

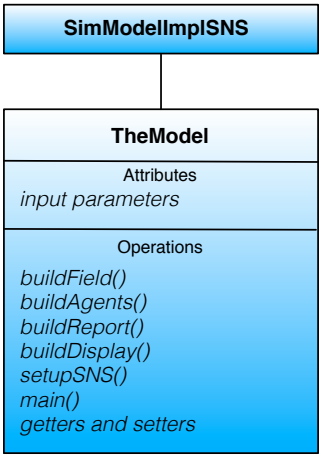


Figure 7.6: *TheModel* class outline.

To meet its definition purposes, *TheModel* class creates the environment (the field and phenomenon) and the particular set of agents corresponding to the scenario. The *buildField()* method just creates a simple default field, as the corresponding behavioural specifications are defined in the *RiverPollutantPhenomenon* class. This *phenomenon* is added to the *field* to incorporate its functionality. The number and kind of agents desired are created within the

buildAgents() method. The monitoring ability of the simulation is specified through the *buildReport()* method, that creates and enables the *reports* of interest. And eventually, *buildDisplay()* prepares drawable elements to appear in the GUI.

The global view of the system's simulation components provided by this class favours a certain application configuration. As a consequence, the kind of relationship established between some elements in the simulation can be specified in the *setup()*. For the precise scenario modelled, the programmer establishes a position relationship between the agent representing the sink and the rest of agents monitoring the state of the environment. As it has already been mentioned in previous sections, COSA requires each node (agent) in the system to be aware of the sink location. Moreover, the sink needs to know the network deployment, that is, the identity of each of the surveillance nodes. We assume that the corresponding information is hard-codified in the elements involved. We use *TheModel* class and its *setup()* method to inform all surveillance nodes about the sink position, and we also provide the sink with a list of surveillance nodes' identities. The value of the unique identifier associated to each agent and its components is also set during this process. Moreover, all the simulation input parameters related to elements' configuration, such as *field dimensions, devices consumption, number of agents,* etcetera are specified through this class. Therefore, *TheModel* class contains *getter* and *setter* methods for each of them.

The *main()* method running all the system belongs to the *TheModel* class. This method creates and loads an instance of this *TheModel* with the values corresponding to the elements' parameters. To conclude, and recalling the design objectives of the platform, this class eventually starts a simulation that has been built from the integration of different interdependent elements.

7.5 Experiments

The description of the software architecture designed to test COSA finishes with the implementation of the classes presented in the previous section. Although all the classes composing the application and the simulation platform have already been defined, running a simulation still needs the specification of input parameters. The particular values given to these parameters will be presented together with the experiments' introduction. The experimental design has been conceived to test COSA's characteristics and to point out strengths and weaknesses in achieving the overall goals.

7.5.1 Hypotheses

The inspiring concept for COSA definition looks for a trade-off between the energy expenditure and the quality of the information received at the server. Grouping the agents and allowing some of them to save energy by not working periodically, causes unavoidably information loss. Nonetheless, the potential improvement offered by the algorithm encourages its use. To test this premise,

the performance of the algorithm is evaluated in different scenarios. The features and the statements that we want to check are:

- COSA increases lifetime average of nodes.
- COSA diminishes uncertainty of the nodes' values.
- COSA and derived strategies reduce error at the cost of network lifetime.

Four kinds of experiments have been performed to test these statements. The first two experiments types focus on basic COSA definition, whereas the other two also deliver information about COSA strategies.

7.5.2 Experiments general framework

The nodes of the network are responsible for monitoring the river's condition and informing the sink about their observations. They are formed by a CPU, battery [Libelium, 2012b], sensor [Libelium, 2012a] and radio [Libelium, 2012c]. These components are modelled after Waspmote devices, real wireless sensors as shown in Figure 7.7. The device specifications are summarised in Table 7.1.

Figure 7.7: Waspmote device [Libelium, 2012b].

Nodes can implement different sampling policies. Regardless of the sampling approach taken by the nodes, each active node has to send its collected data to the sink node. This sink node represents the central monitoring station to which the nodes deployed along the river are reporting to. Differently from sensing nodes, the sink node does not take samples from the environment, neither it is constrained to low power or low processing capacity, as it acts as a server part of the network control unit. Figure 7.8 shows a sample scenario composed of the sink (red circle) and a set of evenly distributed sampling nodes (black circles). Two contamination sources can also be identified, each of them affecting a different number of nodes with different intensities.

The system primary functioning consists of monitoring the environment through periodical collection of samples. The way nodes deployed in the scenario

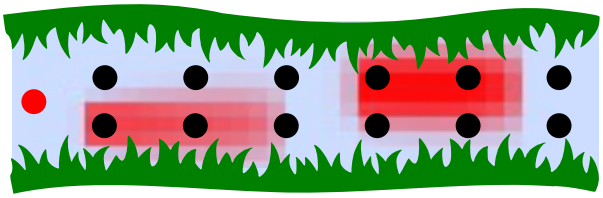


Figure 7.8: Example of a two nodes' grid distribution network.

Table 7.1: Node components specifications

Component	Specification
Battery capacity	13000mAh@3.7V
CPU active consumption	9000uAh@3.3V
CPU sleep consumption	62uAh@3.3V
CPU hibernate consumption	1uAh@3.3V
Radio transmission consumption	210000uAh@3.3V
Radio reception consumption	80000uAh@3.3V
Radio sleep consumption	60uAh@3.3V
Radio bandwidth	156Kbps
Radio Sensing radius	1.5km
Sensor consumption	6uAh@3.3V
Sensor sampling time	1.63s
Sensor gaussian noise	$\mathcal{N}(0, 0.024)$

behave to satisfy this purpose is what defines the applied sampling policy. Regardless of the sampling policy implemented by the nodes, the sink node always acts as the collector that receives the information sampled from the environment.

Random policy represents the base case of all the experimentation to be performed. The Random setting presents a set of nodes (called Random nodes) that take a sample from the environment at a random moment within the sampling period specified for the network and just transmit it to the sink. Nodes implementing COSA use the collected data to establish relationships among them. These relationships are defined according to the functions and parameters presented in Chapter 3.

Table 7.2 presents the values of COSA's parameters used in the experiments. These values indicate the programmer temporal preferences regarding data collection and node activity, as well as the admissible looseness for agents relationship. We study the behaviour of this particular instance of COSA. Changes on these values modify the way agents relate and also the impact of the algorithm in the agent behaviour. Exploring other parameters settings is part

Table 7.2: COSA's parameters values

COSA parameters	Value
d_{max}	1.75
σ_{min}	0.0005
σ_{max}	6
Sampling frequency	10min
Node sleep time	1day
Death threshold	$1{\cdot}10^9$Ans

Table 7.3: Node actions' time cost

Actions	Time
Inner message sending	1 ns
Measure reading	100 ns
Neighbour information update	10 ns
Observed variable model update	10 ms
Adherence evaluation	10 ms
Lead Value evaluation	30 ms

of my future work. An automatic tuning of COSA's parameters to determine optimal behaviour on a particular environment could be achieved heuristically with a Genetic Algorithm, for instance. Regarding the time a node takes to perform typical COSA tasks and inner communication, see Table 7.3 for the values estimated and used in the experiments. Other values might be valid as well, but they would represent a different physical node. Finally, the parameters that characterise the river movement and the nodes' position vary depending on the experimental scenario. Therefore, they are introduced together with the corresponding scenario in the following sections.

Evaluation criteria

The evaluation criteria proposed for COSA experimentation arises from the features that inspired the algorithm conception, which are system energy and quality of the data collected. The system energy is evaluated according to the global available energy of the system in time and the median of this magnitude. Regarding the quality of the information, we propose two assessment criteria: the error registered by every alive node and the entropy associated to the system.

The error measurement represents the deviation of the information available

at the sink for each node from real phenomenon values —as we are simulating, we know the exact value of the phenomenon at any instant. It is evaluated as the addition, for every alive node, of the difference between the phenomenon value known by the sink for each node and the real phenomenon value at the specific nodes' location in the environment. The corresponding mathematical expression can be seen in Equation 7.2, in which $N^t \subseteq A$ represents the set of nodes that have not depleted their batteries at time t; xs_i^t, the value registered by the sink for the observed phenomenon at the position of node i at time t and, finally, xp_i^t is the real value of the observed variable in the time instant t at the position where node i is situated.

$$e^t = \sum_{i \in N^t} \| xs_i^t - xp_i^t \| \tag{7.2}$$

Opposite to the error evaluation, the system's entropy measure takes into account the whole set of agents deployed in the environment, whether they are dead or alive. It is computed as the addition of the entropy value associated to every node in the network. To evaluate the entropy associated to each node, we recover the initial assumption of the phenomenon following a Normal distribution. Therefore, the entropy associated to a general node i can be calculated as shown in Equation 7.3, according to [Goldman, 2005].

$$H_i = \ln\left(\sigma_i \sqrt{2\pi e}\right) \tag{7.3}$$

The entropy measurement establishes a gauge for the available information in the network during the whole simulation time. In this case, the information entropy corresponding to a node i increases as time passes since it last data report. When a node depletes its battery, and it is no longer able to sample its surroundings, the entropy value associated to it increases up to a maximum level that corresponds to a situation of no knowledge at that point —complete ignorance would equate to a flat distribution with a very large σ. We model this process with a time decay function over σ. Equation 7.4 shows the corresponding $\sigma_i(t)$ function.

$$\sigma_i(t) = \begin{cases} \sigma_{bot} & \text{if } t = t_i \\ \sigma_{bot} + \frac{e^{t-t_i}}{e^{t_{max}}} \cdot (\sigma_{top} - \sigma_{bot}) & \text{if } t \neq t_i \end{cases} \tag{7.4}$$

where t_i is the time instant of the last value received from node i; σ_{bot} is the variance of the gaussian noise that the simulator adds to each sensor reading; and, $\sigma_{top} = 100\sigma_{bot}$ represents a very large variance that models maximum ignorance, i.e. a flat distribution. The parameter t_{max} is set to three times the sampling period. Receiving no information from a node for this amount of time would mean a node failure or serious malfunction.

Just as an example and to illustrate the behaviour of this magnitude, Figure 7.9 presents its evolution for a single node that samples the environment at random times in a period of 150 minutes. When the time between samples is less than 20 minutes, the entropy is almost constant at its minimum value. However,

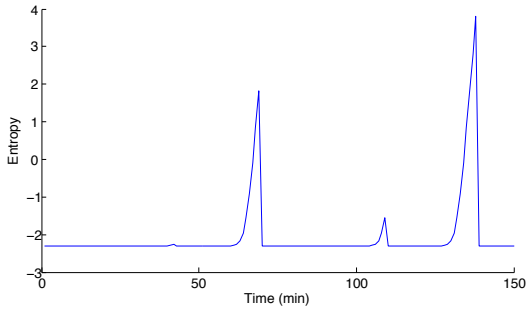


Figure 7.9: Entropy evolution for a single node.

as soon as this time approaches 30 minutes, the entropy value increases, as it occurs in the figure for 70 and 139 minutes time.

The previously presented definitions for error and entropy in the system, allows for an adequate study of the network performance from the quality of information point of view. These measurements, together with the two other energy meters provide with the assessment criteria required. These variables will be evaluated in all the experiments performed.

As previously stated, the set of experiments designed aims at validating the proposed hypotheses regarding COSA performance. The first two scenarios (Scenario I and II) represent stable environments. Scenarios III and IV present more dynamic environments. In both types of scenario, the performance of COSA is tested. The implementation of the Random baseline policy for experimentation requires minimum changes on the agents. Basically, removing the intelligent behaviour implemented in COSA agents.

7.5.3 Scenario I

Scenario I represents the section of a river of 2km by 72.5km. The goal is to detect the presence of hydrocarbon pollutants at any location in the river, therefore, we are interested in monitoring the whole river environment.

The network is deployed to cover all the extension of the river following a regular distribution (see Figure 7.10). The number of nodes considered is set to 50 and their deployment along the river course is assumed to follow a regular chain distribution, in which every node is situated in the middle of the river section and evenly spaced. The position assumed for the sink is also the mid point between the two shores. This assumption does not affect the simulation results. The high number of nodes and the long distances to the sink entail that the communication consumption is not affected by a one-hop distance increase, which is the distance between the mid river point and the shore.

To simulate the river, we define a grid of 10x250 cells whose state is updated every minute. This represents a quite slow river but with an important drag. Values given to river movement parameters can be observed in Table 7.4. The

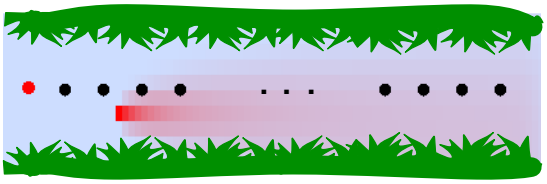


Figure 7.10: Outline of Scenario I.

high ρ value indicates that the drift component governs the river evolution. With respect to the three contributions to this drift component, the central addition dominates over the side ones. Therefore, the simulated river presents a high lineal current.

The rate of pollution sources appearance in this first scenario is quite low. Only three punctual stains of high intensity appear in the system. These stains represent a periodical dump in the river every 5 minutes. The pollutant sources have a random duration between 30 and 60 weeks. They appear at fixed times, specifically at weeks 40, 80 and 100 and at random locations. Hence, they appear separated in time and stay in the system for quite a long time until they are swept away by the river current.

This particular pollution configuration with little punctual stains focus on the behaviour of COSA agents and their capacity to perceive the changes caused by spillages in the environment. We stress the memory of the agents by setting it to a low value as shown in Table 7.4.

In this scenario, we compare the performance of COSA against the baseline Random policy. Agents implementing COSA sample the environment periodically as expected, but instead of directly sending every individual measurement to the sink, these measurements are used to establish peer-to-peer negotiations with neighbours so that sensing coalitions can be formed. Consequently, only one node for each coalition (the leader) senses and sends the information to the sink on behalf of the others, which delegate their tasks on it for a certain period.

The sink node receives the information collected by active nodes. When agents behave according to the random policy, this information corresponds to every single measurement periodically collected by all the random nodes. The sensing task delegation among agents following COSA may cause the sink to receive a group measurement representing the information associated to the set of nodes in that coalition. Assuming a common sample for a set of agents may cause the loss of pieces of information and consequently, the addition of some extra noise to the reported data.

The experimental setup considered is completely defined through Table 7.4. To test if COSA achieves the objectives that inspired its definition, its performance is evaluated in terms of energy consumption and quality of the reported information.

All the experiments have run until every node in the network has completely depleted its battery, at most, 140 weeks in our experiments.

Table 7.4: Scenario I parameters specification

	Parameters	Value
River	Grid dimensions	10x250 cells
	α	0.8
	β	0.1
	γ	0.1
	ρ	0.95
	Update interval	1 min
Stains	Stains dimensions	1x1 cells
	Initial intensity	1
	Pouring intensity	1
	Pouring period	5 min
Agents	Number	50
	Memory capacity	2

Performance analysis in terms of energy

Figure 7.11 and Figure 7.12 depict the behaviour of the system regarding the energy available in the system during the simulation time and the median value of this same magnitude. Particularly, Figure 7.11 shows the ratio of the remaining network energy for both kinds of agent, those behaving according to COSA and those following the Random policy.

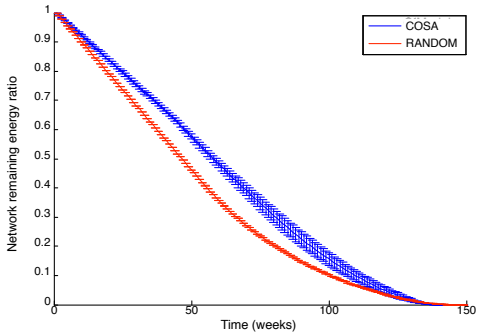


Figure 7.11: Scenario I: Network remaining energy ratio.

In this figure, as initially expected, we can observe how COSA allows the network to keep a higher level of global energy than the Random policy during most of its lifetime, what comes to support the first hypothesis proposed.

However, both sampling policies lead to a very similar network death time.

The energy consumption curve obtained for the Random policy follows a stable pattern, whereas COSA presents more variability, especially by the end of the simulation time. This phenomenon appears due to the different coalition structures that COSA originates in the network over time. The influence of the coalition configurations reached in the network grows as the global energy in the system decreases. Hence, at these middle-end stages, the situation of the leader nodes and the available energy of those nodes still alive have a severe impact on the global energy level.

In contrast, the results obtained for Random nodes clearly show the effect of their independent sampling behaviour. The global energy level is neither affected by their neighbours' state, nor by the dynamics of the environment. Random agents consume energy in doing two tasks: collecting samples from the environment, which requires a quantity of energy that depends on the sampling frequency; and transmission tasks of those samples, which demand an energy expenditure proportional to the square of the nodes' distance to the sink. The decreasing energy curve, therefore, shows the effect of Random nodes dying gradually, depending on their distance to the sink. The worth extension of the useful lifespan of the network can be better identified in the next figure.

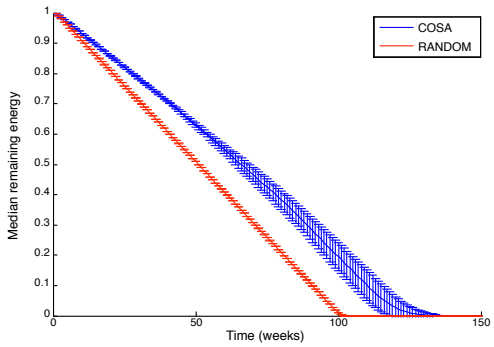


Figure 7.12: Scenario I: Network median remaining energy.

Figure 7.12 represents the median of the nodes' energy values per week. This figure shows that half of the Random nodes are already disabled by week 101, whereas this same value is reached over 30 weeks later for COSA nodes (specifically by week 134). This result translates directly into better system performance during the network lifetime. COSA causes nodes' death to be evenly distributed, which guarantees that the network is going to get a fairly good representation of the whole environment, for most of its lifetime.

This result relates to Figure 7.11, as the nodes that deplete their battery first are the most distant ones to the sink. This situation causes the sink to be blind to this area. Random agents situated far from the sink are the first to die, while the ones situated near the sink hardly spend energy in transmitting. As a consequence, these agents are the last to die. Therefore, in the final stages of

the Random simulation, the network can only sample the sink's surroundings. This situation does not guarantee an adequate surveillance of the subject region. The even battery depletion associated to COSA agents originates a more simultaneous nodes' death phenomenon; although, in terms of global energy in the system, both policies reach zero level at the same time for this first configuration. In this case, this is due to the low and very similar energy costs derived from the actions of nodes situated near the sink, disregarding the behavioural policy selected. That is, the last nodes to die are the closest nodes to the sink, which die at the same time for both policies.

Performance analysis in terms of quality of information

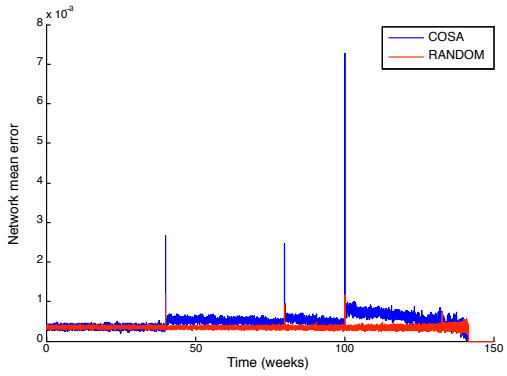


Figure 7.13: Scenario I: Information error at the sink.

The evaluation of the quality of the information bases on error and entropy. Figure 7.13 represents the deviation of the information reported to the sink by Random and COSA nodes. When no pollution phenomenon has appeared yet, both models present the same behaviour. This initial behaviour just shows the effect of the white noise associated to each sensing node. However, as soon as pollution sources start to appear, these reported values also begin to diverge.

The error resulting from the application of the Random policy to the system shows a very stable pattern that only gets altered when a contamination stain appears. As previously explained, three pollution sources have been considered in this simulation for the phenomenon. These stains are of high intensity and appear in the system at random locations but at known times, specifically at weeks 40, 80 and 100. The absolute error value registered at these times reaches approximately $1x10^{-3}$. Nevertheless, this error value returns to the white noise value as soon as all nodes reach their next sampling time.

The error curve of COSA nodes shows a stronger effect of the pollution on its shape. The deviation values reached for the first and second stain are, approximately, of $2.5x10^{-3}$ and the highest error in this case appears for the third stain ($7.25x10^{-3}$). This peak is caused by the characteristic grouping scheme of COSA. The error increase reflects the existence of a leader whose

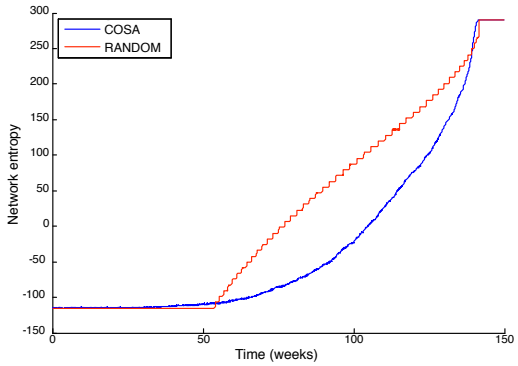


Figure 7.14: Scenario I: Network entropy level.

conditions do not match its followers' situation any longer. As a consequence, this leader transmits samples to the sink that do not represent the coalition members anymore. Regarding the Random policy approach, the moderate error increase with the third stain appearance and at the end stages of the simulation reflects the deviation caused by rapid nodes' death at these late stages.

Both models provide quite good representations of the phenomenon but, as expected, COSA agents provide data that is a little more deviated from reality. However, as Random agents begin depleting their batteries and becoming unable to sense, the uncertainty in the system rapidly increases.

Figure 7.14 shows how the entropy value associated to the network deteriorates almost at a constant pace since the first node's exhaustion for Random policy. At this point, the even battery depletion produced by COSA policy reveals its benefits for the whole system working time. COSA can guarantee a better surveillance of the scenario as it provides a higher level of information during the network lifetime in comparison to Random. Although both models reach the highest entropy value concurrently at the last node's death time, COSA keeps a lower entropy level during the network lifetime, what supports the second hypothesis elaborated in Section 7.5.1.

7.5.4 Scenario II

Scenario II presents the case of a WSN deployment in an already identified polluted area in the river. In contrast to Scenario I, this situation does not require the monitoring of the whole course of the river, but only a particular section that may be located near an industrial site, for instance. We imagine that after pollution, we deploy the sensors in the river. The experiments corresponding to this second scenario focus on the behaviour of the network when all nodes are situated together and distant to the sink. This configuration emphasises the transmission costs and group reunion. The problem area is located at the end of the river course and covers the whole width of the river section an a length of

Table 7.5: Scenario II parameters specification

	Parameters	Value
River	Grid dimensions	10x250 cells
	α	0.8
	β	0.1
	γ	0.1
	ρ	0.95
	Update interval	1 min
Stains	Stains dimensions	15x18 cells
	Initial intensity	1
	Pouring intensity	1
	Pouring period	1 min
Agents	Number	30
	Memory capacity	2

17.5 km. The network deployed to monitor this area is composed of 30 nodes. These nodes are evenly (horizontally and vertically) distributed in the interest area forming a grid of 3 nodes per row (see Figure 7.15).

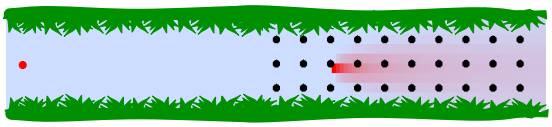


Figure 7.15: Outline of Scenario II.

The river's characteristics regarding its speed and drag conditions are the same as in Scenario I, as it can be observed in Table 7.5. Contaminant sources appear every 35 weeks and last between 6 hours and 5 days maximum. They can appear at any location within the polluted zone where nodes are. Table 7.5 summarises the rest of parameters specifying the pollution characteristics, such as its intensity and extension.

Scenario II represents a different monitoring situation when compared to Scenario I. The analysis of the results obtained from this experimentation will allow for the evaluation of COSA performance in a different situation and also the assessment of the effect of the environment conditions on the algorithm behaviour.

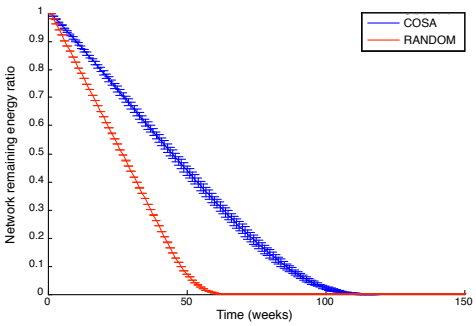


Figure 7.16: Scenario II: Network remaining energy ratio.

Performance analysis in terms of energy

Figure 7.16 presents the network remaining energy ratio for the Scenario II conditions. This configuration returns a quite different result. As in Figure 7.11, it can be appreciated how COSA policy keeps an energy level higher than Random for the whole life of the network. Moreover, and opposite to configuration I, COSA expands the lifespan of this network in 46 weeks. These results support again the first hypothesis proposed.

Another difference between the equivalent results obtained for scenarios I and II refers to the dispersion presented by COSA curve. Whereas Figure 7.11 shows a remarkable growing dispersion in the final stages of the simulation. The dispersion at the end of the simulation is not so notorious in Figure 7.16. The reason for this is the network deployment scheme used in Scenario II. Due to the fact of placing the nodes closer to each other and distant to the sink, the transmission costs of the leaders become somehow standard. Consequently, the energy costs derived from a leader's actions do not differ much whichever node acts as a leader of the coalition. Instead, the number of nodes in a coalition, i.e., the granularity of the network becomes more important, as larger coalitions can be formed.

The same positive effect of nodes' median life increase can also be identified in Figure 7.17. In this case, half of the nodes are already dead 42 weeks earlier when Random model is used in comparison to COSA model. This difference was of 30 weeks in Scenario I configuration, what reasserts the increasing network lifetime tendency.

Performance analysis in terms of quality of information

Measuring the error and the entropy level presented by the network during its lifetime is also essential to understand the effects of the behavioural policy implemented. Figures 7.18 and 7.20 depict the data collected during the simulation referring to quality of the information.

Figure 7.18 shows the addition of the error committed by alive nodes in the network. The times at which the different pollutants appear in the environment

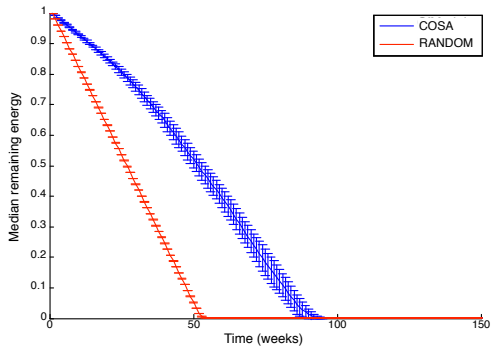


Figure 7.17: Scenario II: Network median remaining energy.

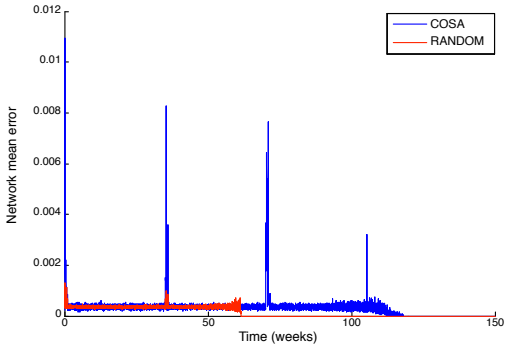


Figure 7.18: Scenario II: Information error at the sink.

can be clearly identified in Figure 7.18 as they cause an important increase in the system's general deviation. This deviation appears for agents implementing either one of the two considered behavioural policies. However, and, as usual, the error registered when pollutants appear is higher for agents implementing COSA.

The high deviations registered by COSA nodes at weeks 0, 35, 70, 105 and 140 are caused by the grouping scheme characteristic of COSA. This scenario configuration especially favours the formation of big coalitions and consequently, higher errors. On the other hand, as Random nodes always sample periodically, the error measurement that they deliver is just caused by the mistaken perception an individual node assumes until its next sampling time. Hence, the error committed by Random nodes reaches approximately the same maximum value for the two scenarios studied.

A detailed view of 7.18 makes relevant the self-adaptation and organisation capacity conferred by COSA to the network. See Figure 7.19, which is a zoom in Figure 7.18. At week 35, when the second pollution stain appears, all nodes are still alive for both behavioural models. Figure 7.19 shows clearly the detection of

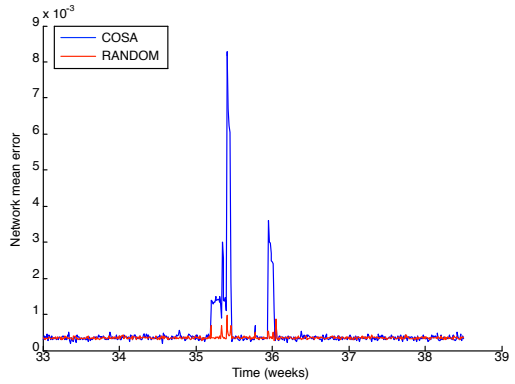


Figure 7.19: Scenario II: Information error at the sink (zoom in Figure 7.18).

the init and end times of the contaminant source. The error returned by nodes implementing COSA is higher than the one obtained for Random nodes at these transitions. However, both behavioural models return to the white noise error value once the stain presence and disappearance have been identified.

One main advantage derived from COSA application refers to the lifespan extension. This achievement allows monitoring the environment longer and, therefore, reporting information when Random nodes are already dead. In Figure 7.18, it can be appreciated how the last two stains appearing in the system are not detected by Random nodes. This loss of information becomes relevant when the information entropy of the system for both models along time is compared (Figure 7.20).

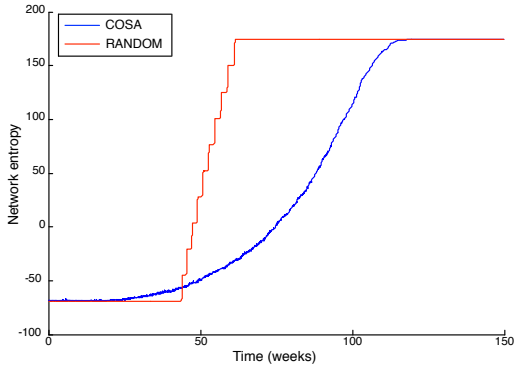


Figure 7.20: Scenario II: Network entropy level.

Figure 7.20 shows how Random policy makes the system maintain its minimum entropy level for almost 44 weeks. From that moment onwards, the entropy of the system increases as each of the nodes dies according to the distance pattern. The maximum information entropy level is reached by week 61. This

situation causes the sink to be blind for the whole scenario as no node is reporting data.

The entropy associated to the network when nodes behaves according to COSA shows a very different pattern. At the beginning of the simulation when all nodes are alive, the entropy curve shows bigger value variations when compared to the Random nodes' graph. This variability reflects the fact of nodes waking up and sleeping according to COSA coalition formation processes. Nevertheless, the even battery depletion of COSA nodes induces a gradual and slower increase in the entropy level of the system, which at the same time reverts in obtaining valid information from the scenario for a longer time. This result supports the second hypothesis stating that COSA diminishes the uncertainty of nodes' values.

This scenario shows the particular COSA trade-off, as this algorithm causes the network to provide information for longer periods at the cost of making bigger errors when compared to a Random sampling regime. However, the maximum deviation reached for either model is on the order of magnitude of $1x10^{-3}$, which represents a pretty low error not causing excessive distortion.

The experiments accomplished show how COSA outperforms Random in terms of energy and entropy level of the nodes implementing this policy. Either the results derived from the Scenario I or II support the proposed hypotheses about COSA behaviour. Hence, we can state that COSA's features fit with its desired characteristics at design phase.

The following experiments focus on COSA strategies to test the third hypothesis "COSA and derived strategies reduce error at the cost of network lifetime." COSA strategies were conceived to improve COSA performance in terms of error, especially when the strategy is applied to a WSN deployed in a highly dynamic environment. COSA strategies definition increases the leaders' activity. This definition itself implies a detriment of COSA main benefit, energy saving. Scenarios III and IV define experimental setups to test how these strategies affect the evolution of control variables, that is, energy, error and entropy of the system.

7.5.5 Scenario III

The network deployment required for Scenario III mimics the chain distribution of Scenario I, as it can be observed in Figure 7.21. As in that experiment setup, the WSN has to monitor the whole river course. Differences between both scenarios refer to the dynamics of the river and pollutant sources. The river presents the same size and speed for all the experiments performed for different scenarios. Nonetheless, the different dynamics of the spillage associated to this scenario comes with a different river flow too. Specifically, the river presents a stronger drag behaviour that corresponds to the higher ρ and α values presented in Table 7.6.

The pollutant phenomenon considered in this scenario appears as a unique intensity-oscillating stain near the sink. Its spewing pace follows a sine function that completes a cycle every 2 hours. The pollutant source lasts for the

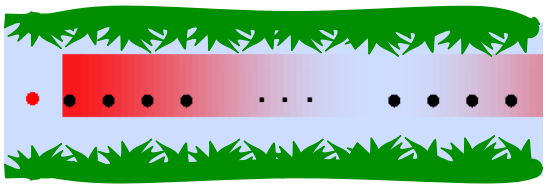


Figure 7.21: Outline of Scenario III.

whole simulation time and spreads along the river due to the river flow. The maximum intensity reached by this stain and other characteristics can be observed in Table 7.6. The pollutant features demand a change in COSA agents' characterisation. As the stain lasts in time and describes a sinusoidal wave, the agent does not need to be so reactive. Hence we increase the memory capacity of the COSA agent. Regarding COSA strategies, the agent behaviour associated to COSA *Sampling Frequency* is predefined with the policy adoption. However, COSA *Coherence* strategy requires the specification of the *coherence threshold*. We tuned this threshold and with high values we found the best network performance.

Table 7.6: Scenario III parameters specification

	Parameters	Value
River	Grid dimensions	10x250 cells
	α	0.9998
	β	0.0001
	γ	0.0001
	ρ	0.9999
	Update interval	1 min
StainSin	Stain dimensions	3x2 cells
	Intensity	1
	Period	2 hours
	Duration	150 weeks
Agents	Number	50
	Memory capacity	10
	Coherence threshold	0.95

The goal of the set of experiments run in this Scenario III is to test the behaviour of COSA and its strategies and also, to compare the results delivered

by each of them. To reach this objective, four sets of simulations are run. Each set compares the behaviour of one COSA approach to Random policy, the baseline already selected for previous experiments. Then, the performance of COSA and Random policy, the performance of COSA *Sampling Frequency* strategy (COSA-SF) and Random, the performance of COSA *Coherence* (COSA-C) and Random and, finally, the performance of the two strategies together, COSA *Sampling Frequency* and *Coherence* joint (COSA-SF+C) and Random policy are evaluated. For each of these experiments, graphs similar to those shown for Scenario I and II referring to energy, error and entropy are obtained. The values assigned to the particular strategy parameters can be identified in Table 7.6.

Apart from the individual strategy study, the comparison of the results delivered by each of them is also an interesting task. To accomplish this goal, and in order to avoid the point by point comparison for the whole simulation time, comparison criteria are established. The election of the comparison points or intervals has been carefully done to guarantee the correctness of the comparison and the general character of the conclusions drawn.

Analysis of results

Before introducing the comparison criteria, we briefly review the behaviour of COSA and its strategies.

Figures 7.22 to 7.24 shows the evolution in time of the evaluation variables (error, energy and entropy). These figures represent the network performance in time.

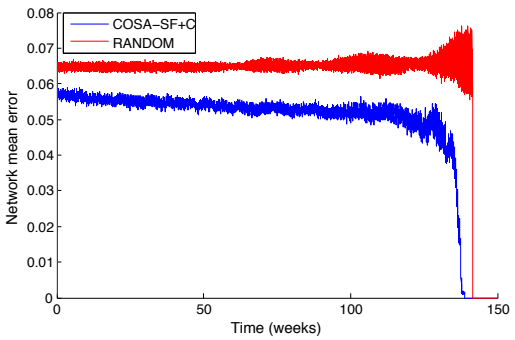


Figure 7.22: Scenario III. Information error: COSA-SF+C and Random.

Figure 7.22 represents the error registered when agents implement the Random policy and COSA-SF+C strategy. The pollutant phenomenon has a periodical behaviour and exists for the whole simulation time. This circumstance causes higher errors when compared to Scenarios I and II, in which the appearance of pollutant sources occurred at specific times. The mean error value associated to the Random policy remains quite constant while all agents are

alive. The corresponding curve associated to COSA-SF+C situates above the Random one. It also shows more variability due to the coalitions' configuration and reconfiguration processes.

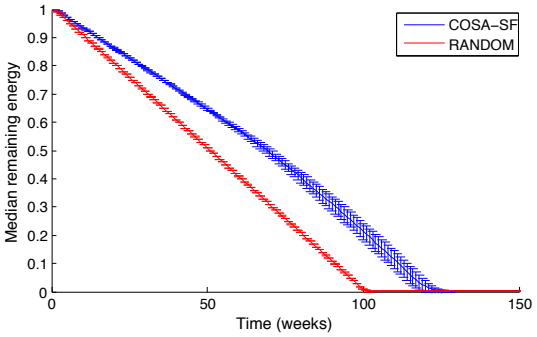


Figure 7.23: Scenario III. Median remaining energy: COSA-SF and Random.

Figure 7.23 shows the median of the nodes' energy values per week. In this case, the strategies compared are Random and COSA-SF. This figure shows that half of the agents deplete their batteries by week 101 when using the Random policy, whereas this situation is reached more than 25 weeks later for COSA-SF. Moreover, COSA-SF allows the network to keep a higher level of global energy than the Random policy during most of its lifetime. This strategy also supports the first hypothesis referring to the energy level of nodes in the system. Regarding the deviation of these results and, as it happened in the analogous Scenario I, this figure shows how the Random policy results in a pattern with constant variance. On the contrary, the increasing variance observed in the curve corresponding to COSA-SF shows the effect of the different energy consumption demanded by leader nodes depending on their position.

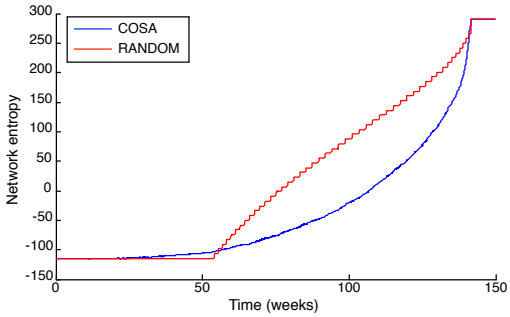


Figure 7.24: Scenario III. Network entropy level: COSA and Random.

Figure 7.24 represents the overall entropy for basic COSA and Random. The Random approach causes the entropy to deteriorate almost at a constant pace

since the first node's battery depletion. The level of entropy when using COSA is lower (i.e. better) during the whole system lifespan. In spite of the high dynamic environment, COSA provides an evenly distributed agents' battery depletion. Once again, this exhaustion scheme allows the network to offer a fairly good representation of the whole environment for most of its lifetime. Hence and, as expected, the uncertainty of the nodes composing the network is reduced when agents behave according to COSA.

These sample figures just show how COSA, its strategies and possible combination of them, when applied to a highly dynamic environment, adjust to the expected behaviour of the algorithm expressed in the hypotheses section. As mentioned before, to compare the performance that each of this alternatives delivers, we define comparison criteria. The identification of an adequate comparison criterion relies on the individual analysis of each policy and the figures previously described.

Comparison of strategies

To compare the results obtained when the WSN deployed in this scenario implements different alternatives of COSA, we take into account the performance in terms of energy and quality of the information. The energy evaluation comparison bases on the analysis of the median value of the energy of the nodes. We focus on this energy perspective as it emphasises the distinctive COSA feature of increasing the node energy level. Regarding the quality of the information, we compare the strategies performance from the error point of view and the entropy perspective.

To compute the gain in terms of error, the difference of the mean error value registered for the COSA policy considered and Random is evaluated. For this calculation, we only take into account the error reported during 100 days of simulation. To select these days of interest, we consider the median of the Random nodes' energy, and we take 100 days around the time when this variable reaches 0.5 value. This constraint is introduced to guarantee that, at the comparison interval time, all the nodes in the network are out of the bootstrapping phase and still alive, hence reporting information. In this period, both sampling policies are in the same conditions, and the comparison is, therefore, fairer. Otherwise, any of the COSA algorithms would be much better in terms of error simply because, in general terms, nodes live longer than those implementing Random policy.

To compute the gain in energy consumption and in entropy we choose a particular point of reference to evaluate the difference in performance. For the gain in energy consumption, the time instant at which the median of the agents' energy value reaches zero is selected. This timestamp is interesting as it represents the moment at which half of the agents in the network have depleted their batteries.

The reference point to evaluate the difference in entropy is set to the point where the overall entropy reaches zero. This gain value indicates how long, in percentage, an algorithm needs to 'loose' information, i.e. to increase the entropy

until reaching zero.

Figure 7.25 summarises the percentage gain obtained by COSA and its strategies with respect to Random sampling.

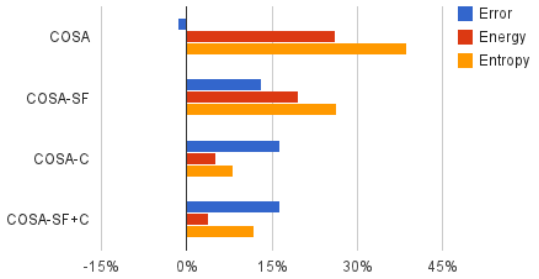


Figure 7.25: Scenario III. COSA gains w.r.t. Random Sampling.

The vertical axis of Figure 7.25 identifies the particular instance of COSA: no strategies (COSA), sampling frequency strategy (COSA-SF), coherence strategy (COSA-C) and both strategies (COSA-SF+C).

COSA shows the expected trade-off between energy consumption and error. As it can be appreciated in Figure 7.25, COSA causes the sink to have slightly higher errors (1.6%) than the Random policy. This loss is however compensated by a gain of 26% in terms of energy consumption and of 39% about entropy.

The first set of bars appearing in Figure 7.25 corresponds to the gain of COSA algorithm compared to Random sampling. It clearly shows that the adoption of COSA policy by the sensing nodes originates a little loss in the accuracy of the information but also, an important increase of the WSN lifespan. This lifespan extension translates into a significant improvement of the quality of the information. The increase in the quality derives mostly from the fact that agents live longer. Hence, the extension of the lifespan of the network does not only represent a reduction of its battery replacement costs but also an improvement of the system's performance.

The results obtained when we used COSA with the *Sampling Frequency* strategy are slightly different. In this case, we get an important improvement in the error gain (reaching a value of almost 13.23%). This improvement comes at the cost of more moderate gains in terms of energy savings and entropy (correspondingly, values of 20% and 26%). Increasing the sampling frequency of the leaders allows them to better follow the changes in the environment caused by the sinusoidal pollutant, therefore, committing less error. However, this extra effort in sampling and transmitting originates also lower gains for the energy and entropy when compared to basic COSA, although the values obtained are still significantly high.

The gain values resulting from the implementation of COSA with the *Coherence* strategy correspond to the third set of bars. This strategy outperforms the *Sampling Frequency* in terms of error gain. The poor performance in terms of energy and entropy (with corresponding values of 5% and 8%) are compensated by an error gain of 16.38%. For the highly dynamic scenario considered and, although the coherence threshold was set almost to one, the cost of breaking coalitions and initiating negotiations reduces the improvements in energy and entropy drastically. Nonetheless, coalition dismantlement causes the nodes to sample the environment at the time this happens, what explains the global committed error reduction. Hence, it is quite obvious that this strategy, with the considered configuration parameters, is not the most convenient for the scenario considered. Although it is still better than Random, it only offers a 3 units improvement about the *Sampling Frequency* strategy, while the energy and entropy gains deteriorates around 15 units.

The results obtained for the combination of both strategies (COSA-SF+C) shows how the *Coherence* strategy has a stronger impact on the combination than the *Sampling Frequency* strategy. In this case, the error gain results in almost the same value as the application of the COSA *Coherence* alone. The error and entropy gains also present low values of 4% and 12%. Therefore, the adoption of these two last strategies does not seem very convenient for this scenario, as COSA-SF+C and COSA-C, both give the best performance in terms of error but at the cost of an important reduction in gains corresponding to energy and entropy measurements. The characteristic trade-off of COSA renders its best results for COSA-SF strategy in this scenario which offers good results in terms of error and energy.

A general view of Figure 7.25 supports the third hypothesis about the cost of improving the network performance in terms of error. The implementation of any of COSA strategies by nodes of the network always translates in a detriment of the energy gain obtained by COSA. The improvement in terms of error is also notorious, as it has already been mentioned. The definition of these strategies provides the programmer with a set of tools that allow him to choose the configuration that fits best to his interests.

7.5.6 Scenario IV

The set of experiments performed in Scenario IV pursues the same objective as in Scenario III. The main difference between these scenarios refers to the network deployment. As in Scenario II, the network aims at monitoring a particular zone of the river, specifically, the more distant area to the sink (see Figure 7.26). This configuration increases the transmission, thus it puts stress on the energy consumption.

The environmental conditions referring to the river movement are the same as in Scenario III, that is a river with a strong drift component. The only pollutant source also mimics the behaviour of the sinusoidal stain in Scenario III, but in this case, the stain appears in the middle of the river and at 45 km from the sink. The problem area extends from this point until the end of the

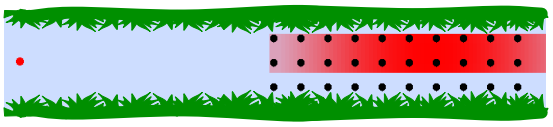


Figure 7.26: Outline of Scenario IV.

Table 7.7: Scenario IV parameters specification

	Parameters	Value
River	Grid dimensions	10x250 cells
	α	0.9998
	β	0.0001
	γ	0.0001
	ρ	0.9999
	Update interval	1 min
StainSin	Stain dimensions	3x2 cells
	Intensity	1
	Period	2 hours
	Duration	150 weeks
Agents	Number	30
	Memory capacity	10
	Coherence threshold	0.95

river section. Nodes are placed in this specific interest area and are deployed as in Scenario II. The network is composed of 30 nodes forming a grid distribution of three nodes per row. The agent characterisation regarding to the behavioural policies implemented coincides with that of Scenario III, as it can be observed in Table 7.7.

The goal of this experimentation is to test the performance rendered by the network when it implements different strategies, and to check the third hypothesis proposed in Section 7.5.1 for this scenario. To meet these objectives, we perform the same set of actions as in the previous section. First, graphical evidence of COSA's characteristics in Scenario IV in terms of energy and quality of the information is presented. Then, gains of each COSA strategy with respect to Random policy are evaluated and compared to find how they adapt to Scenario IV.

Analysis of results

Figures 7.27 to 7.29 show the performance of the network for Scenario IV configuration in terms of error reported by the system, median remaining energy of the nodes and overall entropy level.

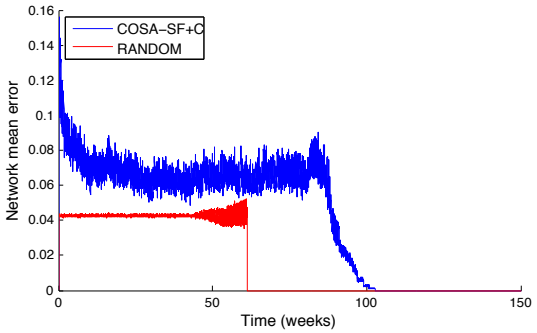


Figure 7.27: Scenario IV. Information error: COSA-SF+C and Random.

Figure 7.27 shows the network mean error per unit time for COSA-SF+C strategy and Random policy. As in Figure 7.22, the reported error by the Random sampling policy shows a stable pattern. This stability is only altered at the final moments of the Random simulation when agents start dying. The specific situation of the nodes far from the sink, together with the characteristic river flow, makes the pollution stain effects smoother, which explains the lower error committed by nodes adopting the random sampling policy. The application of COSA-SF+C strategy, after an initial phase, also returns a quite stable error pattern, although with higher variability around 0.065.

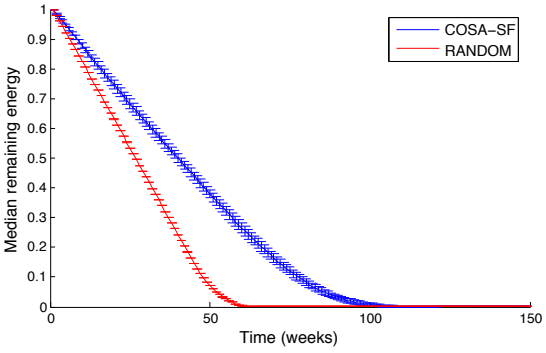


Figure 7.28: Scenario IV. Median remaining energy: COSA-SF and Random.

The median of the remaining energy per node per unit time measures the available energy in the system. Figure 7.28 represents the value of this variable

for COSA-SF and Random strategy. This figure clearly shows the improvement derived from the COSA grouping scheme, as the temporal point when half of the COSA-SF nodes have no energy is reached around 40 weeks later than for Random nodes. This result verifies the first hypothesis proposed about the higher level of energy for nodes implementing COSA when compared to the baseline.

If we compare this figure to its equivalent for Scenario III (Figure 7.23), it can be appreciated that the lifespan of the network for Scenario IV is lower than for Scenario III. In this case, as all the nodes are situated at a considerable distance to the sink, energy demands for information transmission to the server are also higher. The tight grid distribution of the nodes for this scenario implies also low variance in terms of energy, as whichever node acts as a leader, the energy needed to transmit to the sink is almost the same.

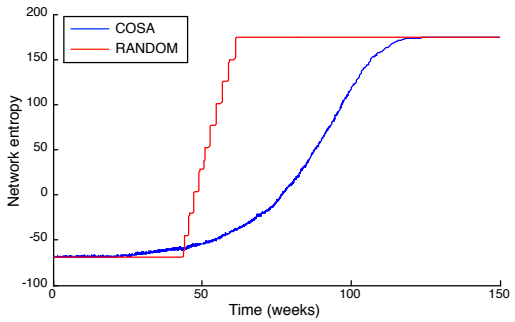


Figure 7.29: Scenario IV. Overall entropy level: COSA and Random.

The strategies considered in this figure are COSA and Random. Both strategies show their typical behaviour pattern. However, the highest (worst) entropy value is reached over 25 weeks later for COSA than for Random. The implementation of COSA policy in this Scenario IV not only guarantees a lower level of uncertainty for the nodes, but also an extension of the network lifespan. This lifetime extension directly implies the availability of information from the environment for longer time.

If we compare this figure to its equivalent for Scenario III (Figure 7.24), we observe that the slope of COSA entropy in Scenario IV is higher than in Scenario III. This effect reflects the different depletion pattern of the nodes' batteries in each scenario.

Comparison of strategies

The results obtained for the application of COSA and its strategies to Scenario IV show a quite different behaviour. The first thing we notice when we observe Figure 7.30 is that none of the COSA strategies reaches a positive gain value for the error measurement. As it occurred for Scenario III, the error gain obtains its worst value for the application of COSA policy and its best one for the combina-

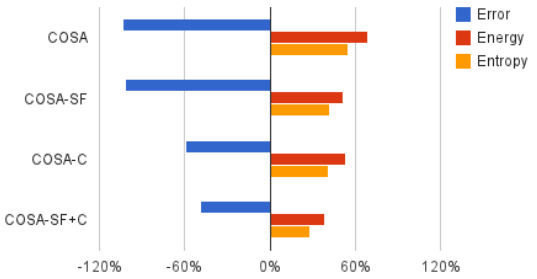


Figure 7.30: Scenario IV. COSA gains w.r.t. Random Sampling.

tion of COSA and its two strategies (COSA-SF+C). The energy measurement shows the opposite behaviour giving the best result for COSA policy and the worst for COSA-SF+C. The relationship between the energy and the entropy measurement also changes in this scenario compared to the previous one. In this case, the entropy gain obtained for any COSA strategy is always lower than its corresponding energy gain. This phenomenon appears due to the specific network depletion pattern, as we will explain later.

In this scenario, with a network composed of 30 nodes situated far from the sink, the error gain obtained when agents implement COSA strategy has a value of -103.3%. That is a loss of 100%, which states that the application of COSA doubles the error committed by the nodes compared to the Random sampling scheme. Therefore, favouring the formation of bigger coalitions in this scenario implies sending information to the sink on behalf of nodes that are poorly represented by their leaders. On the other hand, this high loss in error comes with high values for the energy and entropy gains. The same grouping phenomenon originates high energy savings that render an energy gain of 69.31% and an entropy gain of 55.31%.

The gain values obtained for the adoption of COSA with *Sampling Frequency* strategy shows a very little improvement in terms of the error gain and also, a little detriment of the energy and entropy gains. The error loss is still over 100% (specifically 101.58%), while the energy gain reduces its value to 51.92% and the entropy to 41.98%. Hence, increasing the sampling frequency of the leaders is not very useful for this scenario.

As occurred for the previous Scenario III, COSA *Coherence* strategy causes an important improvement in the error loss, almost dividing its value by 2. The error gain for this strategy and this scenario reaches a value of -59.77%. On the other hand, energy and entropy gains reduce their value, but they do not suffer a dramatic decrease. In this case, the corresponding energy gain is 53.94%, and the entropy gain is 40.83%. The significant reduction in error, together with the low

loss in energy and entropy, show that quick coalition reconfiguration processes allow the nodes to find better distributions and to represent the environment more faithfully.

Finally, the combination of both strategies results in the highest error gain (-49.14%) and the lowest energy and entropy gains (38.8% and 28.27% correspondingly). Once again, the trade-off between energy consumption and accuracy of the information reported to the sink appears for every COSA strategy. For this scenario, COSA is able to extend the network lifetime over 60% compared to the Random sampling strategy at the cost of admitting twice the error reported. The most suitable strategy depends on the programmer preferences.

The results discussed and summarised in Figure 7.30 support again the last hypothesis proposed in Section 7.5.1. COSA strategies support its design conception, as their use can reduce the error committed. Nonetheless, this improvement comes at the cost of the energy available in the network.

7.5.7 Conclusion

The first part of this chapter describes the top layer of the implementation tasks required for experimentation execution. The Riversim module contains the software of the simulation environment of the use case. The connection of this module to the COSA-able WSN is done through the definition of classes that relate both modules. These classes complete the simulation environment and their instantiation corresponds to different experiments.

The goal of the experimentation is the verification of the hypotheses that inspired COSA. The four experiments presented study the behaviour of COSA and its strategies in a river domain. The influence of COSA implementation is compared to the WSN naïve behaviour for two different monitoring problems. These problems consider two different scenarios of the river domain. One of them aims at monitoring the whole course of the river while the other one focuses on a particular problematic area of the waterway.

The results derived from the experiments corresponding to Scenario I and II confirm that COSA behaves as expected. That is, it favours the extension of the nodes average lifetime at the same time that the level of uncertainty in the whole network decreases. Scenarios III y IV define scenarios more dynamic than the previous ones in order to test COSA and its strategies response under these circumstances. The analysis of the results obtained clearly shows the characteristic COSA trade-off between energy expenditure and quality of the information.

A deeper characterisation of COSA performance for different configurations would be desirable. The existence of this aforementioned characterisation, together with a structured identification of environmental features, would allow for the evaluation of the algorithm adequacy in different environments. This analysis would favour COSA testing in new domains, and the exploitation of the implementation work accomplished. The software architecture created for this initial experimentation can easily be adapted to other experimental settings and domains thanks to its modularity and scalability features.

Chapter 8

Conclusions and Future Work

The application domains of WSNs are continuously growing thanks to the development of ever smaller sensors that present more capacities. The improvement of these devices' features and their price decrease have contributed to their widespread use. The special characteristics of WSNs guaranteed by their constituent sensor nodes make possible their deployment in many different areas that could not be monitored with traditional techniques due to difficult access to the zone, the need of high investment or the requirement of a distributed approach. WSNs are an interesting field of study whether it is to improve the network performance and its potential or to exploit the information that they can obtain. WSNs constitute a framework that gathers research questions posed from different communities, such as Electronics, Telecommunication or Software.

One of the major concerns about WSNs refers to their energy management. Sensor nodes are typically constrained in terms of communication capacity, processing and energy as they run on portable batteries. Moreover, depending on the conditions of their deployment environment, battery connection to the power net or replenishment cannot be possible. As a consequence, the limited lifetime of the network can be an important drawback depending on the application domain. This situation has gained the attention of researchers that try to improve different aspects of the WSNs. Different approaches considered for this problem try to improve the hardware design, to upgrade the operation of the nodes' components or to define network energy conservation strategies.

Scenarios in which energy administration is a crucial activity are aquatic environments. In the last years, the application of WSNs to urban environments has gone up tremendously for numerous purposes. The SmartCity concept has become very popular to refer to sensor networks' use in urban areas. In general, these environments allow for part of the nodes connection to the power net and broadband communication. Thus, the main inconveniences derived from the use of a WSN are avoided to a certain extent. These conditions are not reproducible

when the environment to monitor is a river or a lake, for instance. In this case, the use of devices especially adapted to the hard conditions of aquatic environments is necessary. When monitoring a waterway, part of the network deployment can rely on the signalling elements in the river. These environments can be greatly benefited by a WSN implantation. WSNs allow for the extraction of many environmental data that can be used for very different purposes, from ambient conditions monitoring to water state surveillance. This information can be used to attain an adaptive lighting system or to guarantee a certain level of quality of the water by fast identification of changes in its composition. These are the scenario and the problem that have motivated this dissertation.

The addressed problem meets the need of the development of energy-saving strategies for WSNs whose objective is to collect information efficiently from a waterway in order to identify the appearance of contaminant sources in the water. The Multiagent Systems paradigm inspires the proposed strategy. This approach allows for the design of an energy-saving network algorithm grounded on the local activity of the nodes, the *Coalition Oriented Sensing Algorithm*. Formalisation and modelling of this distributed and dynamic problem allowed for the identification of different aspects of the problem that were included in COSA definition.

COSA aims at extending the lifespan of the WSN while guaranteeing a good performance of its goal. Agents (nodes) form coalitions in order to look for a trade-off between the accuracy of the sensed data and their energy consumption. The scenario and algorithm dynamics entail a variable distributed sampling of the environment, in which different number of samples are sent to the sink depending on the instantaneous network configuration. COSA specifies the behaviour of the individual agents to accomplish their tasks of sensing the environment and transmitting this information to the sink and also of reaching an appropriate group configuration. The achievement of a good coalition structure configuration relies on the agent local information about its environment and neighbouring nodes. COSA establishes how this information has to be used and to what extent it affects the agent's behaviour. Thus, this information, together with an appropriate COSA configuration leads to the formation of groups of nodes that act as a single entity, avoiding redundant sensing and transmissions efforts, therefore, saving network energy.

As a consequence, *Coalition Oriented Sensing Algorithm* endows a network with self-organisation capacity. This ability can be used to adapt energy consumption to changes in the environment and, at the same time, to fulfil sampling objectives in terms of the quality of the information reported to the sink.

The study of the algorithm behaviour and its computing properties have been performed over a novel simulation platform, RepastSNS. The election of this platform was principally motivated by its nature. The fact of being a typical MAS simulation platform that included a basic definition of a sensor network components perfectly fitted our demands. However, its novelty and the lack of experimental developments over it turned up in some inconveniences that had to be fixed. Thus, the function of the platform scheduler, which is the

main piece of the platform engine, was amended; erroneous behaviour of some components was fixed, etcetera. These tasks were accomplished without altering the definition principles of the platform in order to not affect its right features. As a consequence, the improvement work developed has resulted in a functional sensor nodes' simulation platform based on MAS.

The general character of RepastSNS promotes its use for different kinds of problems, but it also demands a higher effort for the development of applications. For the particular problem studied in this dissertation, we have designed a layered structure so that each layer adds certain functions to the simulation platform.

The ECA-WSN module is in charge of attaching elementary functions to the node. The energy and communication management strategies provided by this module allows for an easy administration of the node energy. The implemented model relies on a self-management battery and the normalisation of every element consumption. Regarding the node's communication module, it is also adapted to this energy policy and includes the possibility of transmitting both, broadcast and unicast messages. The focus of this module focus on the node's physical attitudes allows for its reuse by other applications, as long as the nodes used for those applications fit these attributes. Analogously, COSA implementation in a network composed of another kind of devices with different physical properties just requires the replacement of this layer.

The COSA-able WSN addresses the embedding of COSA into the agent behaviour. This task is accomplished in a modular way distinguishing the different parts of the algorithm and their effects over the node's components. The modular perspective used for the implementation allows for an easy enhancement of the algorithm. The work performed in this module shows how two different COSA strategies can be implemented just by conducting minimum changes. This feature together with the tuneable character of COSA encourage the proposal of algorithm extensions.

The Riversim module defines the physical environment of the application domain. As it happened with the previous modules, changing the simulation domain strictly affects this module and linking classes. The connection between COSA-able WSN and Riversim modules is fixed through the definition of the specific sensors monitoring the environment and the network deployment. Thus, the implementation work done has resulted in a general development framework that can be used for the creation of different applications.

The new version of RepastSNS platform and the packages corresponding to COSA experimentation performance can be downloaded from http://www.iiia.csic.es/∼delgado/COSA.

Finally, the analysis of the results obtained from the simulation have revealed the utility of local co-ordination when monitoring local phenomena. When a global and dynamic phenomenon affects the network, it causes rapid changes in every node. In these cases, the implementation of local co-ordination strategies, such as COSA, is not appropriate. The lack of neighbouring sensors' samples coincidence, together with the fast changing pace prevent from successful nego-

tiations. Moreover, if a negotiation sporadically successes, this originates high errors due to the quick variation of the phenomenon values. Similarly, the energy saving capacity is highly conditioned by the network topology as the possibility of relationships establishment depends on the number of neighbours and its situation.

In summary, the **contributions** of this dissertation are:

- the proposal of the *Coalition Oriented Sensing Algorithm*.
- A modular architecture and implementation of COSA.
- The development of a workable version of RepastSNS.
- A generic and reusable software model of the physical behaviour of a sensor node.
- Insights on the relationship between local co-ordination and energy saving.

With these propositions in hand, the concepts underpinning COSA and the simulation work performed, we can define the following **future work** lines:

1. Complete evaluation of COSA. The performance of this task requires an exhaustive exploration of the parameters space. This study will generate the layout of the impact of COSA's parameters on the overall network performance under different circumstances. This pattern will fully characterise the algorithm and will allow to establish guidelines on how to use it on different monitoring scenarios.

2. COSA's basic model modifications. The COSA and the associated agents' behaviour rely on the interpretation of local information by means of two functions: *adherence* and *leadership*. Simple changes that could derive interesting results are:

   - To abandon the assumption of Normal behaviour for the observed phenomenon and to evaluate the algorithm performance for different models. This task would entail the establishment of adequate metrics to adapt the *adherence* formulation.
   - To evaluate the similarity factor between two agents using the *Earth Mover's Distance*. Changing this metric would allow for a comparison independent of the variable's model assumed.
   - To focus on individual samples and evaluate the adherence as a simple function of the difference between the nodes' collected data.

3. Network nodes modification by model enhancement. The layered structure clearly separates different aspects of the nodes and the environment. This modular approach permits an easy changing of the physical device modelling. Some changes that can be done are, for instance:

- To consider the *Low Power Listening mode* (LPL) for the nodes when they adopt the *follower* role. This technique allows the radio to lower its consumption in idle states. Its adequacy to COSA should be assessed taking into account the consumption derived from turning on/off the radio each time a state change happens.
- To enhance the sink actuation capacity. COSA conceives the sink as a mere information collector and evaluator. The introduction of a central control unit modelled by the sink can be interesting to increase the self-management capacity of the network. The sink would represent this control unit. This element can take advantage of the information collected by the nodes to act on COSA's parameters. This action would generate a noteworthy feedback phenomenon between the environment conditions and the algorithm particularisation. For instance, stable conditions in the environment could make the sink increase the permissiveness of nodes' association. Changes in COSA configuration initiated by the sink would propagate downstream to the nodes in the network.
- To adopt a realistic routing technique. Once COSA had been fully characterised, an interesting task would be to evaluate its performance when a valid routing strategy is used. The study of how COSA grouping strategy can complement some of the typical clustering algorithms revised at the beginning of the dissertation could deliver important results. For instance, if the COSA leader coincides with a cluster leader, then energy savings would increase as sampling actions would be efficiently performed and information routing would also be optimised. This kind of results would encourage the use of COSA as an appropriate WSN management tool.

Bibliography

[Abbasi and Younis, 2007] Abbasi, A. A. and Younis, M. (2007). A survey on clustering algorithms for wireless sensor networks. *Computer Communications*, 30(14-15):2826–2841. Survey.

[Akyildiz et al., 2002] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114.

[Anastasi et al., 2009] Anastasi, G., Conti, M., Di Francesco, M., and Passarella, A. (2009). Energy conservation in wireless sensor networks: A survey. *Ad Hoc Networks*, 7(3):537–568.

[Bai and Zhang, 2008] Bai, Q. and Zhang, M. (2008). *A Fuzzy Logic-Based Approach for Flexible Self-Interested Agent Team Forming*, volume 89 of *Studies in Computational Intelligence*, chapter Rational, Robust, and Secure Negotiations in Multi-Agent Systems, pages 101–113. Springer Berlin / Heidelberg.

[Bandyopadhyay and Coyle, 2003] Bandyopadhyay, S. and Coyle, E. J. (2003). An energy efficient hierarchical clustering algorithm for wireless sensor networks. In *Proceedings of IEEE INFOCOM 2003*, pages 1713–1723.

[Barton and Allan, 2007] Barton, L. and Allan, V. H. (2007). Methods for coalition formation in adaptation-based social networks. In Klusch, M., Hindriks, K. V., Papazoglou, M. P., and Sterling, L., editors, *CIA*, volume 4676 of *Lecture Notes in Computer Science*, pages 285–297. Springer.

[Basagni et al., 2013] Basagni, S., Naderi, M. Y., Petrioli, C., and Spenza, D. (2013). Wireless Sensor Networks with Energy Harvesting. In *Mobile Ad Hoc Networking: The Cutting Edge Directions*, IEEE Series on Digital and Mobile Communication, chapter 20, pages 701–736. John Wiley and Sons, Inc., Hoboken, NJ.

[Bicocchi et al., 2012] Bicocchi, N., Mamei, M., and Zambonelli, F. (2012). Self-organizing virtual macro sensors. *ACM Trans. Auton. Adapt. Syst.*, 7(1):2:1–2:28.

[Chen et al., 2005] Chen, G., Branch, J., Pflug, M. J., Zhu, L., and Szymanski, B. (2005). *Advances in Pervasive Computing and Networking*, chapter Sense: A Wireless Sensor Network Simulator, pages 249–265. Springer US.

[Chen-Khong and Renaud, 2005] Chen-Khong, T. and Renaud, J.-C. (2005). Multi-agent systems on sensor networks: A distributed reinforcement learning approach. In *Intelligent Sensors, Sensor Networks and Information Processing Conference, 2005. Proceedings of the 2005 International Conference on*, pages 423–429.

[Conte and Castelfranchi, 1995] Conte, R. and Castelfranchi, C. (1995). *Cognitive and social action*. UCL Press.

[Cordina and Debono, 2009] Cordina, M. and Debono, C. J. (2009). Maximizing the lifetime of wireless sensor networks through intelligent clustering and data reduction techniques. In *Proceedings of the 2009 IEEE conference on Wireless Communications & Networking Conference*, WCNC'09, pages 2508–2513, Piscataway, NJ, USA. IEEE Press.

[Dang and Jennings, 2006] Dang, V. D. and Jennings, N. R. (2006). Coalition structure generation in task-based settings. In Brewka, G., Coradeschi, S., Perini, A., and Traverso, P., editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 210–214. IOS Press.

[DARPA, 2013] DARPA (2013). The network simulator ns-2. `http://www.isi.edu/nsnam/ns//`.

[Dyo et al., 2010] Dyo, V., Ellwood, S. A., Macdonald, D. W., Markham, A., Mascolo, C., Pásztor, B., Scellato, S., Trigoni, N., Wohlers, R., and Yousef, K. (2010). Evolution and sustainability of a wildlife monitoring sensor network. In *SenSys*, pages 127–140.

[Elkind et al., 2013] Elkind, E., T., R., and N.R., J. (2013). *Multiagent Systems*, chapter Computational Coalition Formation. MIT Press, Cambrige, MA.

[Endesa, 2014] Endesa (2014). Smartcity málaga. `http://www.smartcitymalaga.com`.

[Gasser, 1993] Gasser, L. (1993). Social knowledge and social action: heterogeneity in practice. In *Proceedings of the 13th international joint conference on Artifical intelligence - Volume 1*, pages 751–757, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Gaston and desJardins, 2005] Gaston, M. E. and desJardins, M. (2005). Agent-organized networks for dynamic team formation. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 230–237, New York, NY, USA. ACM.

[Glinton et al., 2008] Glinton, R., Scerri, P., and Sycara, K. (2008). Agent-based sensor coalition formation. In *Information Fusion, 2008 11th International Conference on*, pages 1 –7.

[Goldman, 2005] Goldman, S. (2005). *Information Theory*. Dover Phoenix Editions.

[Griffiths and Luck, 2003] Griffiths, N. and Luck, M. (2003). Coalition formation through motivation and trust. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, pages 17–24, New York, NY, USA. ACM.

[Heinzelman et al., 2000] Heinzelman, W. R., Chandrakasan, A., and Balakrishnan, H. (2000). Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8 - Volume 8*, HICSS '00, pages 8020–, Washington, DC, USA. IEEE Computer Society.

[Horling and Lesser, 2004] Horling, B. and Lesser, V. (2004). A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.*, 19(4):281–316.

[IIIA-CSIC, 2012] IIIA-CSIC (2012). Repast sensor network simulation toolkit. `http://www.iiia.csic.es/~mpujol/RepastSNS/`.

[Korkalainen et al., 2009] Korkalainen, M., Sallinen, M., Kärkkäinen, N., and Tukeva, P. (2009). Survey of wireless sensor networks simulation tools for demanding applications. In *Proceedings of the 2009 Fifth International Conference on Networking and Services*, ICNS '09, pages 102–106, Washington, DC, USA. IEEE Computer Society.

[Kraus et al., 2003] Kraus, S., Shehory, O., and Taase, G. (2003). Coalition formation with uncertain heterogeneous information. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, pages 1–8, New York, NY, USA. ACM.

[Kumar et al., 2011] Kumar, V., Jain, S., and Tiwarei, S. (2011). Energy efficient clustering algorithms in wireless sensor networks: A survey. *International Journal of Computer Science Issues*, 8(2):259–268.

[Lasassmeh and Conrad, 2010] Lasassmeh, S. and Conrad, J. (2010). Time synchronization in wireless sensor networks: A survey. In *IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the*, pages 242–245.

[Lawton, 2003] Lawton, J. H. (2003). *The Radsim*, chapter The Radsim Simulator. Kluwer Academic Publishers.

[Lesser et al., 2003] Lesser, V., Tambe, M., and Ortiz, C. L., editors (2003). *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer Academic Publishers, Norwell, MA, USA.

[Libelium, 2012a] Libelium (2012a). Sensor board. http://www.libelium.com/es/101651651444/.

[Libelium, 2012b] Libelium (2012b). Waspmote. http://www.libelium.com/documentation/waspmote/waspmote-technical-guide-eng.pdf.

[Libelium, 2012c] Libelium (2012c). Xbee pro 900. http://www.digi.com/pdf/ds_xbeepro900.pdf.

[Manning and Schütze, 1999] Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing*. Massachusetts Institute of Technology.

[Martinez et al., 2004] Martinez, K., Hart, J., and Ong, R. (2004). Environmental sensor networks. *IEEE Computer*, 37(8):50–56.

[Matamoros, 2008] Matamoros, J. M. (2008). Migració d'una plataforma de simulació de xarxes de sensors a repast. Projecte de final de carrera de Enginerya Informàtica. UB.

[Mihailescu et al., 2011] Mihailescu, R.-C., Vasirani, M., and Ossowski, S. (2011). Dynamic coalition adaptation for efficient agent-based virtual power plants. In Klgl, F. and Ossowski, S., editors, *Multiagent System Technologies*, volume 6973 of *Lecture Notes in Computer Science*, pages 101–112. Springer Berlin Heidelberg.

[Mitra and Nandy, 2012] Mitra, R. and Nandy, D. (2012). A survey on clustering techniques for wireless sensor networks. *International Journal of Research in Computer Science*, 2:51–57.

[Omicini, 2000] Omicini, A. (2000). Soda: Societies and infrastructures in the analysis and design of agent-based systems. In *In this volume*, pages 185–193. Springer-Verlag.

[Padhy et al., 2006] Padhy, P., Dash, R. K., Martinez, K., and Jennings, N. R. (2006). A utility-based sensing and communication model for a glacial sensor network. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1353–1360, New York, NY, USA. ACM.

[Pujol-Gonzalez, 2008] Pujol-Gonzalez, M. (2008). Plataforma per a la simulació de xarxes de sensors. Projecte de final de carrera de Enginerya Informàtica. UAB.

[Rahwan and Jennings, 2008] Rahwan, T. and Jennings, N. R. (2008). An improved dynamic programming algorithm for coalition structure generation. In *Proc 7th Int Conf on Autonomous Agents and Multi-Agent Systems*, pages 1417–1420.

[Rebollo et al., 2014] Rebollo, M., Carrascosa, C., and Palomares, A. (2014). Follow the leader in a consensus network as a solution to manage an smart grid: The balearic islands case. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 1655–1656, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

[Rogers et al., 2006] Rogers, A., Dash, R., Jennings, N., Reece, S., and Roberts, S. (2006). Computational mechanism design for information fusion within sensor networks. In *Information Fusion, 2006 9th International Conference on*, pages 1–7.

[Rogers et al., 2008] Rogers, A., Osborne, M., Ramchurn, S., Roberts, S., and Jennings, N. (2008). Information agents for pervasive sensor networks. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 294–299.

[Ruairi and Keane, 2007a] Ruairi, R. M. and Keane, M. T. (2007a). The dynamic regions theory: Role based partitioning for sensor network optimization. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*.

[Ruairi and Keane, 2007b] Ruairi, R. M. and Keane, M. T. (2007b). An energy-efficient, multi-agent sensor network for detecting diffuse events. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 1390–1395.

[Shehory and Kraus, 1998] Shehory, O. and Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(12):165 – 200.

[Siham and El Ganami, 2012] Siham, A. and El Ganami, D. (2012). Advanced passive clustering-threshold a maintenance mechanism of the cluster structure. *Journal of Theoretical and Applied Information Technology*, 46(2).

[Siham et al., 2013] Siham, A., G., D. E., and Abdelilah, M. (2013). Clustering algorithms based on energy efficiency in wireless sensor networks: survey. *ARPN Journal of Engineering and Applied Sciences*, 8(10):785–795.

[Sims et al., 2003] Sims, M., Goldman, C. V., and Lesser, V. (2003). Self-organization through bottom-up coalition formation. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, pages 867–874, New York, NY, USA. ACM.

[Smith, 1980] Smith, R. G. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *Computers, IEEE Transactions on*, C-29(12):1104 –1113.

[Sobeih et al., 2005] Sobeih, A., peng Chen, W., Hou, J. C., chuan Kung, L., Li, N., Lim, H., ying Tyan, H., and Zhang, H. (2005). J-sim: A simulation environment for wireless sensor networks. In *In Annual Simulation Symposium*, pages 175–187. IEEE Computer Society.

[Sourceforge, 2012] Sourceforge (2012). Repast agent simulation toolkit. `http://repast.sourceforge.net/repast_3/`.

[Srisooksai et al., 2012] Srisooksai, T., Keamarungsi, K., Lamsrichan, P., and Araki, K. (2012). Practical data compression in wireless sensor networks: A survey. *Journal of Network and Computer Applications*, 35(1):37 – 59. Collaborative Computing and Applications.

[Sundani et al., 2011] Sundani, H., Li, H., Devabhaktuni, V., Alam, M., and Bhattacharya, P. (2011). Wireless sensor network simulators a survey and comparisons. *Journal of Computer Networks (IJCN)*, 2:249–265.

[Sycara, 1998] Sycara, K. P. (1998). Multiagent systems. *AI Magazine*, 19(2):79–92.

[Telefónica et al., 2014] Telefónica, I., Alcatel-Lucent, S., Cantabria, U., Ericsson, D., and Surrey, U. (2014). Smart santander. `http://www.smartsantander.eu`.

[Varga and Hornig, 2008] Varga, A. and Hornig, R. (2008). An overview of the omnet++ simulation environment. In *In Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering.*

[Vig and Adams, 2007] Vig, L. and Adams, J. A. (2007). Coalition formation: From software agents to robots. *J. Intell. Robotics Syst.*, 50:85–118.

[Vinyals et al., 2011] Vinyals, M., Rodriguez-Aguilar, J. A., and Cerquides, J. (2011). A survey on sensor networks from a multiagent perspective. *Comput. J.*, 54(3):455–470.

[Yick et al., 2008] Yick, J., Mukherjee, B., and Ghosal, D. (2008). Wireless sensor network survey. *Computer Networks*, 52(12):2292 – 2330.

[Younis and Fahmy, 2004] Younis, O. and Fahmy, S. (2004). Heed: A hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing*, 3:366–379.

[Zambonelli and Omicini, 2004] Zambonelli, F. and Omicini, A. (2004). Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283.