Università della Calabria

Dipartimento di Ingegneria Informatica, Modellistica,

Elettronica e Sistemistica

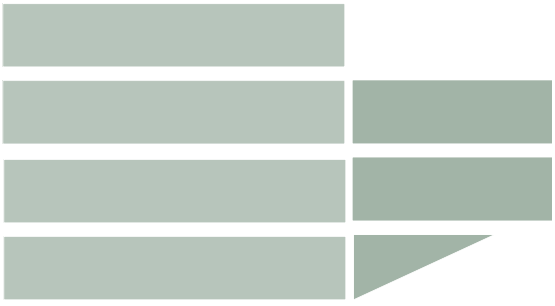
Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica

XXVI Ciclo

Tesi di Dottorato

Autonomic Computing-Based Wireless Sensor Networks

Stefano Galzarano






UNIVERSITÀ DELLA CALABRIA

Dottorato di Ricerca in

Ingegneria dei Sistemi e Informatica

XXVI Ciclo

Tesi di Dottorato

Joint-PhD con la Eindhoven University of Technology (TU/e)

Con il contributo della Commissione Europea, Fondo Sociale Europeo

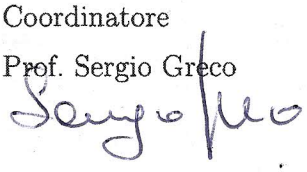
e della Regione Calabria

Autonomic Computing-Based Wireless Sensor Networks

Stefano Galzarano

Coordinatore

Prof. Sergio Greco



Supervisori

Prof. Giancarlo Fortino

Prof. Antonio Liotta (TU/e)

DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA,

ELETTRONICA E SISTEMISTICA

Novembre 2013

Settore Scientifico Disciplinare: ING-INF/05

Abstract

Wireless Sensor Networks (WSNs) have grown in popularity in the last years by proving to be a beneficial technology for a wide range of application domains, including but not limited to health-care, environment and infrastructure monitoring, smart home automation, industrial control, intelligent agriculture, and emergency management.

However, developing applications on such systems requires many efforts due to the lack of proper software abstractions and the difficulties in managing resource-constrained embedded environments. Moreover, these applications have to meet a combination of conflicting requirements. Achieving accuracy, efficiency, correctness, fault-tolerance, adaptability and reliability on WSN is a major issue because these features have to be provided beyond the design/implementation phase, notably at execution time.

This thesis explores the viability and convenience of Autonomic Computing in the context of WSNs by providing a novel paradigm to support the development of autonomic WSN applications as well as specific self-adaptive protocols at networking levels. In particular, this thesis provides three main contributions. The first is the design and realization of a novel framework for the development of efficient distributed signal processing applications on heterogeneous WSNs, called SPINE2. It provides a programming abstraction based on the *task-oriented paradigm* for abstracting away low-level details and has a platform-independent architecture enabling code reusability and portability, application interoperability and platform heterogeneity. The second contribution is the development of SPINE-* which is an enhancement of SPINE2 by means of an autonomic plane, a way for separating out the provision of *self-* techniques* from the WSN application logic. Such a separation of concerns leads to an ease of deployment and run-time management of new applications. We find that this enhancement brings not only considerable functional improvements but also measurable performance benefits. Third, since we advocate that the agent-oriented paradigm is a well-suited approach in the context of autonomic computing, we propose MAPS, an agent-based programming framework for WSNs. Specifically designed for supporting Java-

based sensor platforms, MAPS allows the development of general-purpose mobile multi-agent applications by adopting a multi-plane state machine formalism for defining agents' behavior. Finally, the fourth contribution regards the design, analysis, and simulations of a self-adaptive AODV routing protocol enhancement, CG-AODV, and a novel contention-based MAC protocol, QL-MAC. CG-AODV adopts a "node concentration-driven gossiping" approach for limiting the flooding of control packets, whereas QL-MAC, based on a Q-learning approach, aims to find an efficient radio wake-up/sleep scheduling strategy to reduce energy consumption on the basis of the actual network load of the neighborhood. Simulation results show that CG-AODV outperforms AODV, whereas QL-MAC provides better performance over standard MAC protocols.

Riassunto

Le Wireless Sensor Networks (WSNs) sono cresciute in popolaritá negli ultimi anni, avendo dimostrato di essere una tecnologia vantaggiosa per una vasta gamma di domini applicativi, inclusi, ma non limitati a, health-care, monitoraggio ambientale e delle infrastrutture, smart home, controllo industriale, agricoltura intelligente, e gestione delle emergenze.

Tuttavia, lo sviluppo di applicazioni per tali sistemi richiede molti sforzi a causa della mancanza di appropriate astrazioni software e delle difficoltá nel gestire ambienti embedded aventi risorse limitate. Inoltre, queste applicazioni devono soddisfare una serie di requisiti tra loro contrastanti. Raggiungere accuratezza, efficienza, correttezza, tolleranza ai guasti, adattabilitá e affidabilitá in una WSN è un importante problema poiché tali caratteristiche devono essere fornite ben oltre la fase di progettazione/implementazione, specialmente in fase di esecuzione.

Questa tesi esplora la fattibilitá e la convenienza dell'Autonomic Computing nel contesto delle WSN, fornendo sia un nuovo paradigma per supportare lo sviluppo di applicazioni autonomiche, che specifici protocolli di rete autoadattativi. In particolare, questa tesi fornisce tre contributi principali. Il primo riguarda la progettazione e la realizzazione di un innovativo framework per lo sviluppo di efficienti applicazioni per signal processing distribuito su WSN eterogenee, chiamato SPINE2. Esso fornisce un'astrazione di programmazione basato sul paradigma *task-oriented* per astrarre quelli che sono i dettagli di basso livello ed è caratterizzato da un'architettura platform-independent per consentire la riusabilitá e portabilitá del codice, nonché l'interoperabilitá delle applicazioni e l'eterogeneitá delle piattaforme impiegabili. Il secondo contributo riguarda lo sviluppo di SPINE-*, il quale rappresenta un'evoluzione di SPINE2 in seguito all'aggiunta di un piano autonomico, un modo per separare la fornitura di *tecniche self-** dalla logica dell'applicazione. Tale separazione logica porta ad un facile deployment e gestione a run-time di nuove applicazioni, garantendo non solo notevoli miglioramenti funzionali, ma anche vantaggi prestazionali. Terzo, poiché sosteniamo che il paradigma orientato ad agenti è un approccio che ben si adatta al contesto dell'autonomic computing,

proponiamo MAPS, un framework di programmazione basato ad agenti per WSN. Specificamente progettato per supportare piattaforme di sensori basate su Java, MAPS consente lo sviluppo di applicazioni general-purpose multi-agente utilizzando una macchina a stati finiti multi-piano come formalismo per la definizione del comportamento degli agenti. Infine, il quarto contributo riguarda la progettazione, l'analisi e la simulazione di un'evoluzione del protocollo di routing AODV, CG-AODV, e di un protocolo MAC contention-based, QL-MAC. CG-AODV adotta un approccio "node concentration-driven gossiping" per limitare il flooding dei pacchetti di controllo, mentre QL-MAC, basato su un approccio di Q-Learning, mira ad individuare una strategia efficiente di schedulazione del wake-up e sleep per la radio, in modo da ridurre il consumo energetico sulla base dell'effettivo carico di rete del vicinato. I risultati di simulazione mostrano che CG-AODV supera in prestazioni AODV, mentre QL-MAC fornisce migliori performance rispetto protocolli MAC standard.

Contents

List of Figures

List of Tables

1

Motivation, Objectives and Organization of the Thesis

1.1 Motivation

Wireless Sensor Networks (WSNs) [1] are collections of tiny, low-cost, and low-power devices (sensor nodes) having sensing, computing, storing, communicating and possibly actuating capabilities. Every sensor node is programmed to interact with the other ones and with its environment, constituting a unique distributed and cooperative system aiming at a global behavior and result. WSNs are a powerful technology for supporting a lot of different real-world applications such as health-care, environment and infrastructure monitoring, smart home automation, emergency management. By showing also great potentialities for numerous other different domains [2], they will become in the imminent future a fundamental part of our life, with profound impact on our daily activities.

Unfortunately, many issues still need to be addressed in this field. In this thesis, we focus on the following.

First of all, developing WSN-based applications is a hard work because it implies dealing with many different programming aspects, ranging from low-level management of the sensor nodes hardware and the radio communication to high-level concepts concerning final user applications. Another limiting factor is represented by the difficulties in implementing efficient applications on scarce hardware resources in terms of power supply, computational capability, and memory. Also, without a specific programming supporting tool, developers have to translate the global distributed application into local behavior and functions to be coded into every node-level environment, which may be a very long and error-prone task. Furthermore, with the ever increasing application complexity in terms of functionalities and services provided to the users, the need for integrating different sensor architectures, each of which providing specific capabilities, will lead to further challenges in terms of platforms interoperability in such a heterogeneous environment.

Second, not only the development phase but also the post-deployment one should be been taken into the necessary consideration. In fact, the global qual-

ity of an application not only depends on how well it has been designed and implemented but also on how well it can deal with problems at runtime as well as how well it can configure itself, depending on the interaction with the external environment and other interconnected networks. Moreover, since sensing faults and unforeseen events may happen during the application execution, it is not reasonable for a WSN to be manually and constantly maintained and supervised after deployment. For such reasons, the system itself needs to own specific self-management capabilities. In particular, autonomic characteristics [3] may be incorporated into a WSN so to ensure fundamental properties such as fault tolerance, adaptability, reliability, and maintainability [4].

Third, we are not only concerned with the application logic of a sensor network. In order for a WSN to better adapt to the network status or changing network conditions (and thus preserving good network performance) or to better manage the sensor node resources (e.g. reducing the battery consumption for extending the system lifetime), the underlying communication protocols should also provide some kind of self-adaptive behavior, possibly with very little computational and communication overhead.

1.2 Objectives and contributions

The purpose of this thesis is to present programming frameworks, techniques and protocols conceived for dealing with the above discussed issues, i.e. aiming at supporting rapid development of WSN applications as well as enabling self-management behaviors at runtime.

In particular, the following objectives have been achieved:

- The need for a powerful yet simple software development tool for better exploiting the current and the future WSNs has led to the development of a novel programming framework, named SPINE2 (Signal Processing In Node Environment, version 2), which inherits from its predecessor just its philosophy-of-use and the purposes, but offering a completely different programming abstractions and software architecture (see Section 3.2.8 for a comparison). Intended to be used for a rapid development of effective yet efficient WSN signal-processing applications, it aims at providing developers an intuitive and straightforward design model based on simple graphical constructs. Specifically, a *task-oriented approach* has been chosen as high-level modeling paradigm for defining distributed applications on heterogeneous embedded environments by abstracting away any low-level details concerning specific platforms and communication protocols, for the benefit of code reusability and portability. Since a WSN system usually demands very strict requirements in term of efficiency and stability, the SPINE2 middleware, running on the sensor nodes, has been carefully designed and implemented so to guarantee high-performance execution of the task-based applications on such resource-constrained embedded systems.

Moreover, its platform-independent architecture has been also conceived to allow for an easy and fast porting procedure to support new sensor platforms and thus enabling application interoperability and platform heterogeneity.

- The second contribution refers to a novel autonomic architecture, named SPINE-*, that aims at easily and explicitly defining the autonomic properties without affecting the WSN application behavior and logic. Conceived as an enhancement of SPINE2, SPINE-* inherits all its previously discussed characteristics such as code reusability, application portability and interoperability, and platform heterogeneity. In addition, it extends the conventional developing framework by means of an autonomic plane, a way for separating out the provision of self-* properties from the WSN application logic. This is enabled by the task-oriented paradigm which is well suited for allowing the necessary separation of concerns for independently dealing with both the application plane and the autonomic plane. This separation of concerns leads to an ease of deployment and run-time management of new applications. We find that this enhancement brings not only considerable functional improvements but also measurable performance benefits, as shown in a case study where a specific WSN application has been improved by means of proper self-healing tasks aiming at mitigating erroneous behaviors at runtime, and thus improving the quality of an application, due to the effects of sensor data faults.

- Designing and implementing autonomic behaviors in a computing system do not require a certain approach or a specific programming language. However, many researchers in the Autonomic Computing community have recognized that many ideas developed in the MAS (Multi-Agent Systems) community may likely be fruitfully adopted for enabling the various self-* properties. The employment of decentralized, autonomous but interacting and collaborating entities, defined as a result of the decomposition of complex problems and tasks, clearly prove to be of some benefits also for the autonomic computing. Motivated by this consideration, an innovative agent-oriented programming framework for WSN, namely MAPS (Mobile Agent Platform for Sun SPOT) is proposed. MAPS has been designed for the development of Java-based applications running on WSN based on the Sun SPOT sensor platforms. Specifically, MAPS agents' behavior is modeled by means of a multi-plane state machine formalism driven by ECA (Event-Condition-Action) rules. Performance evaluation of MAPS has been carried out as well as a comparison with other current Java-based MASs for WSNs.

- The fourth contribution regards the definition, implementation and simulation of specific MAC and routing protocols showing self-adaptive capabilities. In particular, a novel contention-based MAC protocol for WSNs,

named QL-MAC and based on a Q-learning approach, is proposed. The protocol aims to find an efficient wake-up strategy to reduce energy consumption on the basis of the actual network load of the neighborhood. Moreover, it benefits from a cross-layer interaction with the network layer, so to better understand the communication patterns and then to significantly reduce the energy consumption due to both idle listening and overhearing. The proposed protocol is inherently distributed and has the benefits of simplicity, low computation and overheads. As for the routing protocol, an enhancement of AODV, called CG-AODV, is proposed. In particular, a "node concentration-driven gossiping" approach is adopted for limiting the flooding of control packets in the standard protocol definition and improving the network performance in terms of packet delivery ratio and path discovery delay. The "node concentration" is introduced as a new measurable quantity and, in contrast to the standard network density, not only take into consideration the spatial distribution of nodes but also their transmission range.

1.3 Organization of the Thesis

This thesis is organized as follows:

- Chapter 2 illustrates an overview on Autonomic Computing by introducing its history, its main characteristics and the research contributions over the last years. Moreover, some applications of the Autonomic Computing principles to the WSN domain are presented, as well as the use of Machine Learning techniques as effective instruments for enabling autonomic behavior in WSN.

- Chapter 3 presents our proposals for enabling the development of autonomic applications on WSNs by means of proper frameworks. It first discusses about the need for proper programming tools for better dealing with the development of effective and efficient WSN applications. Then, it illustrates the SPINE2 programming framework by describing in details its characteristics and comparing it with some related work. Afterward, the autonomic programming framework SPINE-* is presented and its features and architecture described. Moreover, some case studies involving both SPINE2 and SPINE-* are shown. Finally, the use of mobile agents in the WSN domain is discussed and the characteristics of MAPS, its architecture and its agent programming model are delineated.

- Chapter 4 presents specific self-adaptive networking-level protocols for WSNs. In particular, an enhancement of the AODV routing protocol, called CG-AODV, and a novel contention-based MAC protocol, called QL-MAC, are described in details. Moreover, simulation results are shown to

validate their ability to adapt to the network status and guarantee better network performance with respect to other standard protocols.

- Finally, Chapter 5 presents a summary of the main results of this thesis, along with some concluding remarks. Afterward, possible future research works that can derive from the work here presented and a list of the publications related to the thesis are shown.

2

Towards Autonomic WSN

Wireless Sensor Networks (WSNs)[1] have gained attention in the last years thanks to the potentialities offered by the use of small, low-cost, and low-power devices (sensor nodes) providing data acquisition, processing and wireless communication capabilities. The application domains that may benefit from the use of wireless sensor networks (WSNs) have grown rapidly [2]: health-care, infrastructure monitoring, smart home automation, environment surveillance, emergency management, industrial control, precision agriculture and many others.

Specifically, when applied to the human body, such sensor networks are called Body Sensor Networks (BSNs) [5, 6]. BSNs are currently having a significantly increasing research interest thanks to their abilities in enabling continuous and real-time human monitoring at low cost by guaranteeing ease of deployment, fault tolerance, and non-invasive operations. Most of the current systems based on BSNs are related to the health-care domain [7, 8, 9]. Examples of health-care applications include early detection or prevention of diseases, elderly person assistance at home, rehabilitation after surgeries, cognitive and emotional recognition, medical assistance in disaster events. However, wearable systems have great capabilities for different other application domains, which are gaining more and more attention in the last years: e-Social [10], e-Sport and e-Fitness [11, 12], e-Entertainment and interactive computer games [13, 14], and e-Factory [15].

As the cost and the dimension of wireless sensors diminish we should expect in few years to see a large number of heterogeneous WSNs/BSNs deployed to become very important building blocks for supporting the upcoming "Internet of Things" (IoT) [16, 17, 18]. In such a vision, the everyday objects surrounding us will become intelligent and proactive components connected to Internet (Smart Objects) for generating and consuming information and providing better solutions in any kind of sector. Thus, next WSN applications will not only have to be conceived as a stand-alone solution but as piece of a big puzzle where the intelligent interaction and synergy with the environment

along with the Internet connectivity represent the key points for providing the next generation innovative value-added services.

The benefits of connecting both WSN and other IoT elements go beyond the simple data remote access and the potential of such an integration have already been foreseen by several international companies. A notable example, which extends the concepts to generic WSNs, is the "Smarter Planet"[1] by IBM. This projects a strategic vision for building a smarter planet by instrumenting, interconnecting and building intelligence into systems and considering sensors as fundamental pillars in intelligent cities management. Another example is the CeNSE[2] project by HP Labs, focused on the deployment of a worldwide sensor network in order to create a "central nervous system for the Earth".

The importance of WSNs in such complex environments has led to the need for proper techniques and tools capable of effectively WSN applications managing so to guarantee their correctness and efficiency at execution time as well as fault tolerance, adaptability, and reliability. And the *Autonomic Computing* paradigm with its self-* properties can perfectly meet these critical requirements.

2.1 Autonomic Computing

The Autonomic Computing (AC) paradigm [3] has been conceived as a response to the rapidly growing complexity of computing systems characterized by an increasing heterogeneity, distribution, dynamism and pervasiveness. In a scenario where plethora of technologies are tightly connected to each others, operators and administrators are facing more and more complex failures and configuration problems that are becoming hard to be manage except by highly skilled human experts. But manual control is time-consuming, expensive, and error-prone; and since nowadays the cost for IT personnel is exceeding equipment costs, there is an increasing economic need to automate systems maintenance.

As the name may suggest, the autonomic computing vision takes inspiration from the autonomic nervous system which is in charge of managing and regulate all the non-conscious activities of the body such as heartbeat rate, breathing rate, and body temperature. In a similar way, an autonomic system should be able to autonomously take charge of certain system functions so to avoid an active human intervention for managing those activities concerning low-level details and let users to combine their efforts on higher level concerns. As a consequence, the human operator has a new role: defining general policies that guide the self-management process without any need for him to directly control the system.

[1] http://www.ibm.com/smarterplanet/
[2] http://www.hpl.hp.com/research/intelligent_infrastructure/

Several research disciplines have studied many of the issues related to AC for many years. For instance, fault tolerant computing and the need for robust systems are not new, as well as all the researches in the field of Artificial Intelligence and Software Engineering trying to satisfy desired system properties such as reliability, maintainability, adaptability and security. What is new with the AC approach is its way to bring all the relevant research ares together towards an unique research direction aiming to achieve long-standing dependability of complex systems [19].

2.1.1 A brief history

The term Autonomic Computing was first coined by Paul Horn of IBM in 2001, who compared complex computing systems to the human body [20]. In his manifesto, he also advised the need for a management component acting in a similar fashion as the autonomic nervous system. AC was then conceived as a model for computer systems having self-management capabilities, i.e. able to autonomously take care of the regular maintenance and thus reducing the system administrators' workload. Moreover, four different properties were identified for characterizing an autonomic system: self-configuration, self-optimization, self-healing, and self-protecting.

However, the concepts behind self-management has been already investigated few years earlier in a military project.

In the late 1990s, the Defense Advanced Research Projects Agency (DARPA) was involved in a project called *Small Unit Operations-Situational Awareness System (SUO-SAS)*. Its aim was to provide soldiers with a robust, ad-hoc self-forming, and self-managing network to support them in tactical military operations in difficult environmental conditions, where it is impossible to rely on a fixed infrastructure. Once a soldier has turned the radio device on, its software system was able to automatically acquire the correct channel and start exchanging and collecting the status reports with the other devices deployed over the battlefield, as well as getting information from ground sensors and unmanned aerial vehicles, so to be aware of the global situation of the war theater. Radio devices were able to transmit in a wide range of possible frequencies and bandwidth, in order to adapt each communication to the current conditions. For instance, if a soldier was many miles away from the closest one, the radio automatically would switch to low frequencies and, because of the corresponding lower bandwidth, would exchange only the proper and most relevant information. And all this operations would occur without any direct intervention by the soldier.

Another project funded by DARPA, namely *Dynamic Assembly for Systems Adaptability, Dependability and Assurance (DASADA)*, started in 2000. It looked for effective means to monitor large software system performance and feasible adaptation engines capable of dynamically reassemble, removed and replaced software components at runtime with no system outages. Similarly to the IBMs autonomic computing initiative, such technologies were

conceived for dealing with the complexity of distributed software systems and enabling mission critical applications to meet high reliability and adaptability requirements.

In 2004, the DARPA *Self-Regenerative Systems* program aimed to develop techniques for military computing system capable of critical functionality at all time, despite unintentional errors or deliberated attacks. One of these technique was to generate different versions of a software having similar functional behavior but different implementations, such that a potential attack was unlikely to be able to affect all of them at the same time. Also, by directly modifying the binary code, e.g. by pushing randomly sized block onto memory stack, would be harder for a hacker to exploit vulnerabilities on it. Moreover, a scalable intrusion-tolerant architecture and a technique to prevent malicious system operator from initiating an attack were developed.

It is worth to mention here, the strong interest of NASA in autonomic computing which could make possible for a deep-space probe to autonomously and quickly adapt its behavior to extraordinary situations. This is very important when the round-trip delay of communication between the probe and the mission control on Earth is so long that it would be impossible to rapidly send new commands from Earth related to some critical decisions. One of the most interesting space program related to autonomic computing is the *Autonomous NanoTechnology Swarm (ANTS)*, started in 2005, which consists in using hundreds of miniaturized, autonomous spacecrafts to explore Solar System bodies like the asteroids in the asteroid belt. Inspired by insect colonies, they should form small coordinated groups, each of them having specific role in the mission but capable to reconfigure in case of loss of spacecrafts.

2.1.2 The self-* properties

As discussed by Horn in his manifest [20], the main properties of an autonomic computing system are the following [3]:

- *Self-configuration.* The system must be able to configure itself based on specific high-level policies and objectives and to effectively adapt itself on the needs of the user and the platform by dynamically adding, replacing or removing its components with no system outages.
- *Self-healing.* The system must be able to autonomously prevent, detect, and possibly remedy its possible malfunctions to ensure adequate levels of reliability. The nature of possible problems that can be detected is quite broad, ranging from low-level hardware failures to high-level erroneous software configuration. However, it is important that the operations related to the self-healing process do not affect other vital components in the system.
- *Self-optimization.* The system must be able to make better use of the available resources and to plane its activities to constantly pursue the maximization of its performance (proactive behavior).

- *Self-protection*. The system must have the ability to provide specific levels of security by preventing, detecting and remedying any intrusion and malicious attack aimed at its sabotage. Moreover, it should also protect itself from user inputs that may be inconsistent, implausible, or dangerous.

The above listed properties are usually known as *self-CHOP* properties. However, others specific "self-*" properties can exist too. Terms like self-organization, self-inspection, self-repairing, self-monitoring, self-testing and many others have all been used both in academic and industrial research.

2.1.3 The MAPE-K loop

Along with the definition of Autonomic Computing and its fundamental properties, IBM also suggested an architecture as a reference model for an autonomic element [21]. It is usually known as *MAPE-K loop*, an acronym derived from the initials of the main components/activities constituting the model (see Figure 2.1): Monitor, Analyze, Plan, Execute, Knowledge. At some extent, this model is similar to a generic agent model that perceives its environment through sensors and uses these pieces of information to define the actions to be execute.

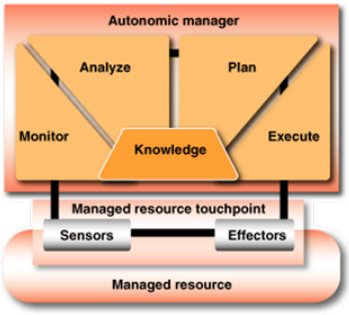


Fig. 2.1. The MAPE-K loop reference model.

The *managed resource* represents any kind of hardware or software system component which is coupled with its specific *autonomic manager* such that it can exhibit an autonomic behavior. It can be, for instance, an operating systems, a database, a web service, a server, a router, a CPU, a network and so on.

The *Monitor* component is in charge of collecting details and properties of the managed resource by means of proper sensors, probes, and gauges. All information that are of significance for the self-* properties are then aggregated, correlated and filtered so to provide a proper state representation of the resource to be analyzed. The *Analyze* phase involves data analysis and reasoning

on the information provided by the monitor. Such analysis is also influenced by the data stored in a knowledge-base. Analysis results are then passed to the *Plan* function, which defines and selects a series of actions that produce the required changes needed to achieve specific objectives and goals. Finally, the selected actions are put in place in the *Execute* phase and, by means of specific effectors, the managed resource behavior changes accordingly. The *Knowledge* component simply represents a generic data storage, containing basic and aggregated information as well as analysis results, shared among all the functions in the loop.

2.1.4 State-of-the-art

A first example of implementation of the MAPE-K loop can be found in the IBM Autonomic Computing Toolkit[3]. It was developed as a reference framework for incorporating autonomic functionalities into software systems but it was not designed to be a complete autonomic manager. It was meant to be used in all the context where the autonomic manager could be implemented at application level and not at more lower levels like the operating system or even the hardware ones. It is implemented in Java but is able to communicate with other applications via XML data exchange. In [22] the Toolkit was used and extended to implement an autonomic network service configuration, where the servers act as autonomic managers whereas the other network devices are modeled as managed resources.

Another well known IBM toolkit is ABLE [23]. Every autonomic manager can be modeled and implemented as an agent or a set of agents (i.e. each single autonomic task can be represented and coded into a specific agent). Thus, the system implemented through this toolkit appears as a multiagent software architecture, whereas the adopted programming language is Java.

Another implementation of a complete autonomic loop, called Kinesthetics eXtreme, can be found in [24]. They tried to address the problem of integrating autonomic properties into legacy systems, i.e. system not originally designed for including autonomic behaviors. Since it is not possible to modify such systems, they propose a way for monitoring their status by means of software "sensor" placed on top of the existing APIs. On the basis of the monitored information, proper adaptation and repair functions can be issued to the system. Specifically, these planned actions are executed by means of a Workflow engine (Workflakes [25]): the autonomic manager deploys proper Worklets (mobile agents) to the controlled legacy system on which a Worklet Virtual Machine is running. Such a virtual machine interprets and executes the agents whose actions are translated into specific commands/actions for the legacy system thanks to proper host-specific adaptors.

With respect to the aforementioned tools, a different approach is to develop specific autonomic middlewares providing self-management functionalities to

[3] http://www.ibm.com/developerworks/autonomic/r3/probdet1.html

the applications implemented on top of them. An example of this approach can be found in [26]. Since the self-* properties are provided to applications running on top of the middleware, differently from the Kinesthetics eXtreme autonomic tool, it is not possible to apply this approach to legacy systems.

Along with the proposals of real implemented autonomic frameworks and tools for building autonomic systems, other research activities provide methodologies for modeling autonomic systems such as in [27], [28], [29], and [30].

When AC is specifically applied to computer networks, research community usually refers to it also as Autonomic Network. Many past and current research works have been conducted for exploiting the basic autonomic principles into traditional networks by following different directions and reaching different levels of success. A rather extensive survey on Autonomic Network Management can be found in [31]. The interest in this research field is proved by several international projects that started in the last few years.

BISON[4] (Biology-Inspired techniques for Self-Organization in dynamic Networks) aimed to develop techniques and tools for building self-organizing, self-repairing and adaptive Network Information Systems by taking inspiration from biological processes.

The ANA[5] project has developed a novel Autonomic Network Architecture able to adapt and re-organize on the basis the users' needs in terms of working, economical, and social requirements.

Haggle[6] is an autonomic networking architecture specifically designed and developed for exploiting opportunistic contacts between mobile users, i.e. guarantee communication even under intermittent network connectivity. conditions.

CASCADAS[7], abbreviation of "Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services", had as an objective the development and the experimentation of an autonomic framework for enabling composition and deployment of an ecosystem of services capable to self-adapt to unpredictable situations.

The EFIPSANS[8] project aims at exploring the ways how appropriate IPv6 protocol and architectural extensions can be exploited for building new autonomic networks and services.

2.2 Autonomic WSN

The autonomic computing is an effective paradigm conceived for facing with the arising needs of managing the growing complexity of computing systems.

[4] http://www.cs.unibo.it/bison/
[5] http://www.ana-project.org/
[6] http://www.haggleproject.org/
[7] http://acetoolkit.sourceforge.net/cascadas/
[8] http://www.efipsans.org/

All the desired properties of an autonomic system perfectly meet the basic requirements of self-managing of an embedded, distributed and highly dynamic computing environment like a WSN: tolerance to faults; adaptation to the loss or the addition of nodes by preserving the application functionality; support for cooperative management of critical resources (energy and computational ones); optimization of sensor node resources; impracticality of recharging or replacing the battery and in general impracticality of maintenance by technicians.

However, by contrast to the more traditional networks, the features of WSNs make the design and the implementation of such management architecture rather challenging. And thus, the autonomic-related branch of research of the sensor networks domain has not been fully investigated yet.

Among the technical challenges that needs to be addressed, one of the biggest ones is the design of an effective network management architecture for continuously supporting WSN applications at runtime.

2.2.1 State-of-the-art

As previously mentioned, unlike research on AC for the traditional and most common networks, research on autonomic architectures specifically conceived for WSNs is not well established yet.

Few examples of system architectures for WSN management can be found in literature: MANNA [32], BOSS [33], WinMS [34], and Starfish [35].

MANNA is a general management architecture for WSN that considers three different abstraction planes in the definition of each management function: functional areas (fault, configuration, maintenance, performance, security, accounting), management levels (business, service, network), and WSN functionalities (sensing, processing, communication). Each function represents the lowest granularity of a management action, whereas set of functions are composed to form high-level management services. Furthermore, the MANNA architecture defines WSN models that represent aspects of the network, i.e. abstract visions of the system and its low-level details, and serve as information for the execution of the management functions.

The BOSS architecture is based on the standard UPnP protocol, which allows devices to be automatically discovered, configured and controlled over a traditional network, without the need for manual configuration. To allow tiny sensor devices with limited resources to use UPnP, a software bridge running over the basestation acts as a mediator and provides sensor network management services, which are required to manage the WSN. Its architecture is composed of different functional components: control manager, event manager, service manager, and a set of sensor network management services.

WinMS is an adaptive policy-based sensor network management system allowing dynamic nodes adaptation to changing network conditions. It provides both local and decentralised management schemes and a global management

scheme, the former according to neighborhood network states, the latter according to global knowledge of the network. Communication is based on a lightweight TDMA protocol, FlexiMAC, that collects network state data and disseminates management information through a gathering tree.

Starfish is a policy-driven framework for self-adaptive sensor networks. It is based on an embedded node-side policy management system, called Finger2, and a desktop client that facilitates definition of strategies for dealing with sensor errors, component failures and reconfiguration requirements. A module library is included for simplifying the programming of motes by providing a high-level definition language for defining both policies and user applications behavior.

Apart from few generic system architectures for self-managing sensor networks, as the ones previously discussed, most of the research efforts that have been carried out so far on autonomic WSN are mainly concerning self-healing and fault management techniques and systems [36, 37, 38, 39, 40] and in particular on the effect of node failures in the network [41, 42, 43, 44]. Although very important in the WSN domain, few works have been devoted to sensor data faults. In [45] a distributed fault detection algorithm is proposed, through which a faulty sensor can diagnose itself through comparing its own sensed data with the average of neighbors data. In [46], an algorithm for faulty sensor identification, based on neighboring coordination is proposed. If the difference of a reading of a sensor is larger than the neighbors median readings, the sensor is likely to be faulty. The authors in [47] discuss the impact of sensor faults in two case studies of body sensor deployment and describe mechanisms that will allow detection of those faults during systems operation.

2.2.2 Introducing intelligence in WSN

Research on autonomic WSN is in its early stage for what concerning effective management architecture devoted to satisfy the self-* properties on the WSN and on the user applications. However, in the last years many researchers have focused their attention to specific aspect of these networks. In particular, in order to address rather challenging issues in such complex and dynamic environments, Machine Learning (ML) approaches and techniques have been successfully employed [48]. Although not explicitly reported in their research papers, such research results can be considered as approaches for satisfying specific self-* properties by combining elements of learning and adaptation. Examples of tasks successfully addressed by ML are data aggregation and fusion, energy aware routing, radio scheduling and MAC, security, optimal deployment and localization. The wireless ad-hoc nature of WSNs, their physical distribution, mobility and topology changes and energy limitations make them so challenging for being addressed by conventional and not “intelligent” algorithms.

The application of various ML techniques to all these specific aspects have been extensively surveyed in [48]. Specifically, the main computational intel-

ligence paradigms taken into consideration by this survey are: Reinforcement Learning, Neural Networks, Swarm Intelligence, Evolutionary Algorithm, and Fuzzy logic.

This survey shows in details which specific algorithm has been mainly employed for each of the aforementioned issues in the literature. Moreover, it also presents authors' personal evaluation on the applicability and suitability of these methods depending on their specific characteristics. For instance, although several papers have been devoted to the adoption of swarm intelligence in routing problems, this is not actually a wise choice. In fact, this ML paradigm requires high communication overhead for sending ants separately for managing the routes and this is not a feasible approach in the WSN domain where the energy waste due to excessive communications should be taken seriously into consideration. On the contrary, reinforcement learning is considered the best option when dealing with distributed and dynamic problems like routing and clustering for WSNs as well as radio scheduling and MAC. From this analysis, it also emerges that neural networks has been rarely used in WSNs though it is a well studied paradigms and there exist already many different models. Finally, one of the authors' main concerns is the lack of proper comparison with conventional protocols representing the state-of-the-art in WNSs in order to better highlight the advantages of using ML approaches. Furthermore, authors also suggest to improve and refine already existing solutions with adapting techniques based on ML so that they can be easily tested in real-world contexts besides simulations.

3

Application-level approaches: middlewares for autonomic WSN

One of the main open challenges in the WSN context is represented by a limited support for application development, since the gap between sensor platforms (both hardware and OS) and final desired user applications is hard to fill and a standard approach has not been defined yet. Designing and programming WSNs are complex tasks, not only due to the challenge of implementing complex applications on heterogeneous devices having very constrained resources (computational, storage and energy), but also due to the need for guaranteeing their correctness even in case of problems at runtime [49, 50].

Thus, it is clear the importance of development tools integrating proper features for both abstracting away low-level programming details and defining self-management capabilities at application level.

In this chapter, WSN programming approaches are first briefly discussed. Then, novel frameworks aiming at supporting rapid development of WSN applications as well as enabling self-management behaviors at runtime, SPINE2 and SPINE-*, are described in details. Both conceived for the development of distributed signal processing applications, their main characteristics are the use of task-oriented paradigm as a programming abstraction and a platform-independent architecture which enable code reusability and portability, application interoperability, and platform heterogeneity. SPINE-* also allows to define specific self-* behavior without affecting the original application logic. Moreover, an agent-based programming framework, namely MAPS, and its modified lightweight version, TinyMAPS, are described. Specifically conceived for Java-based sensor platforms, the their agents behavior is modeled by means of a multi-plane state machine formalism driven by ECA (Event-Condition-Action) rules.

Some case studies related to these frameworks are also presented for demonstrating their validity and effectiveness.

3.1 Developing WSN applications

A WSN application can be basically built accordingly to one of the following development approaches [51]: (i) platform-specific programming, (ii) framework-aided programming, and (iii) automatic code generation.

Platform-specific programming involves the use of a certain API (Application Programming Interface) provided by a specific sensor platform (e.g. by its operating system) and resulting in complex, low-level programs that specify the behavior of individual sensor nodes. Thus, applications are expressly developed to be tailored for a specific purpose and optimized for achieving high performance. Developing applications directly on top of the operating system primitives does not allow for a rapid and effective application development, since programming individual node's behavior in a distributed context through a low-level programming language is an error-prone and time-consuming task. In fact, developers need to directly interface to the network and node resources and have to explicitly deal with messages transmission and parsing, sensor readings, events and interrupts handling. In general, sensor network operating systems, like other embedded systems, tend to leave more hardware control to developers than any other computational system. Thus, they have to cope with device drivers, scheduling problems, code optimizations and others low-level matters. A comparison among different sensor platforms (specifically, TinyOS[1] [52], MANTIS[2] [53], and the Ember ZigBee implementation[3]) when considering them from the application developers perspective, can be found in [54]. For small and simple applications these are not relevant problems, but as application complexity arises they become a strong limiting factor. And this is particularly true in the light of the fact that many recent application domains demand for using multiple interconnected sensor networks that need complex multi-platform applications to be managed. Furthermore, in the future we will see more and more growing claims for the so called "Internet of Things" which is a large vision of a human being's life daily supported by pervasive computing systems [55]. On the basis of the previous considerations, there is a strong interest in using powerful software instruments that simplify application development on WSNs.

Framework-aided programming makes use of specific tools in order to simplify and speed up the application development process. Usually, a programming framework comes with both some kind of programming abstractions provided to developers and a proper software (its middleware) able to support the actual "execution" of such abstractions. In particular, such a middleware is a software layer usually residing on top of the sensor platform architecture (both hardware and OS) and in charge of "translating" the high-level constructs used by developers into the actual running distributed application.

[1] http://www.tinyos.net
[2] http://mantisos.org
[3] http://www.ember.com

The middleware is properly designed to take care of all the necessary low-level management routines so supporting the abstractions by controlling the node resources, covering up constraints and providing the necessary high-level services to application developers. In few words, a framework is conceived to provide new capabilities for an effective and efficient sensor data extraction, manipulation, and transport, so that it could be possible for a programmer accessing to a more intuitive programming interface for high-level WSN operations such as data collection and aggregation, signal processing and event notification. The actual problems with such a high-level programming approach are that (i) it is very difficult to provide a unique application development tool capable of effectively supporting all potential applications for WSN, as WSNs are application-specific networks, and also (ii) user applications can suffer from poor performances at runtime if the middleware is not well optimized.

For such reasons, *domain-specific frameworks* are starting to obtain an increasing interest. By combining the best characteristics of application-specific programming and framework-aided programming, they not only include libraries and tools that can be easily reused for different applications of a certain domain but also they can be optimized for achieving the best performance over such a domain. BSNs represent a good example of a more specific domain inside the WSN one, in which the primary requirements for a framework is to provide signal processing intensive tasks on wearable nodes.

The *automatic code generation* approach consists in making use of tools capable of generating low-level code for a specific target platform starting from a well-defined application model. So, similarly to other programming frameworks, these tools allow developers to specify an application by means of proper high-level abstractions. The only difference relies on the actual code running on each node, since they do not come with a common middleware running on the sensor nodes. In fact, the high-level model is directly translated, by means of a proper code generator, into a source code ready to be installed on a certain sensor platform. Thus, for each supporting platform a specific "translator" needs to be provided. While this approach, also known as model-based design, represents a standard approach for several domains, such as automotive electronics, its application in the WSN one has not been widely investigated yet and a very few tools have been actually proposed and tested.

3.1.1 Development approaches comparison

Table 3.1 summarizes the main characteristics of the above described approaches. It is quite obvious how the use of a programming framework can increase the development effectiveness if compared to building application through low-level programming languages and platform-specific APIs. The code efficiency remains the best characteristic of a custom application, but at the cost of long development and debugging time, which can be highly reduced

by means of dedicated tools offering a rapid prototyping process through high-level constructs and application modeling languages.

Table 3.1. Comparison among different WSN programming approaches.

	Platform-specific programming	General purpose framework	Domain-specific framework	Automatic code generation
High-level application modeling		X	X	X
Rapid prototyping		X	X	X
Ease of debugging		X	X	X
Quick application deployment		X	X	
Application reconfiguration at runtime		X	X	
Code efficiency	X		X	X
System interoperability		X	X	
Support to specific application needs			X	

If a fast application deployment and a runtime reconfiguration features are some of the desired requirements, then the best choice is to rely on the frameworks providing a distributed middleware running all over the sensor network. Usually, once the middleware has been installed on the nodes constituting the WSN, developers do not have to physically access them anymore, since their applications (which, we remind, are properly described by a specific abstract model) are simply loaded over the network in a distributed way by means of proper radio messages. Thus, the middleware is in charge of managing the reconfiguration and the upgrade of the user applications. As a consequence, the WSN maintenance time is also greatly reduced. On the contrary, both the low-level programming method and the code generator-based one do not provide such features, since they produce a firmware that needs to be manually uploaded on each single node, unless the physical sensor platform itself provides an over-the-air (OTA) programming functionality (nowadays, very few platforms come with such a feature).

Another important feature is represented by the system interoperability, i.e. the possibility for different applications, even if running on different sensor platforms, to easily collaborate. Programming frameworks can straightforwardly provide inter-operation among applications because they are able to communicate by means of the same set of proper high-level messages. Anyway, in order to allow applications to operate in a heterogeneous environment, it is quite obvious that the different sensor platforms need to employ the same low-level communication protocol. Developers relying on platform-specific or

code-generation programming approaches, instead, have to put much more efforts and time in order to achieve similar results.

In few words, such a comparison points out that, among the described approaches, the domain-specific one can offer a wide range of benefits: while guaranteeing high efficiency, it allows for a more effective development of customized applications. And this is provided with little or no additional hardware configuration by means of high-level programming abstractions specifically tailored for a specific reference application domain.

3.1.2 Brief frameworks overview

As previously discussed, for addressing WSN programming issues and supporting developers in a fast and effective application development, in the last decade many frameworks and tools have been proposed, each of which differing on the programming constructs and the abstract application model.

In the following, the existing programming abstractions and related reference frameworks are reported and described.

Database model: (TinyDB [56], Cougar [57], SINA [58]). The database model lets users view the whole sensor network as a virtual relational distributed database system allowing a simple and easy communication scheme between users and network. Through the adoption of easy-to-use languages, the users have the ability to make intuitive queries for extracting the data of interest from the sensors. The most common way for querying networks is making use of a SQL-like language, a simple semi-declarative style language. This model is mainly designed to collect data streams, with the limitation that it provides only approximate results. Also, it is not able to support real-time applications because it lacks time-space relations between events.

Agent-based model: (MAPS [59], AFME [60], Agilla [61], SensorWare [62], actorNet [63]). The agent-based programming model is associated with the notion of multiples, desirable lightweight, agents migrating from node to node performing part of a given task, and collaborating each other to implement a global distributed application. An agent could read sensor values, actuate devices, and send radio packets. The users do not have to define any per-node behaviors, but only an arbitrary number of agents with their logics, specifying how they have to collaborate for accomplishing the tasks needed to form the global application on the network. Middleware according to this model provides users with high-level constructs of a formal language for defining agents characteristics, hiding how collaboration and mobility are actually implemented. The reasons in adopting such a model is mainly due to the need for building applications that can be reconfigured and relocated. Moreover, the key of this approach is that applications are as modular as possible to facilitate their distribution through the network using mobile code.

Domain-specific approach: (SPINE [51, 64], Titan [65], RehabSPOT [66], BMF [67], CodeBlue [68]). Domain-specific frameworks are in the middle be-

tween application-specific code and general-purpose middleware approaches. They specifically address and standardize the core challenges of WSN design within a particular application domain. While maintaining high efficiency, such frameworks allow for a more effective development of customized applications with little or no additional hardware configuration and with the provision of high-level programming abstractions tailored for the reference application domain. In particular, SPINE is a lightweight and flexible framework providing specific libraries and tools that can be easily reused for developing signal processing intensive applications on BSN, whereas Titan is specifically designed to perform context recognition in dynamic sensor networks by representing data processing through reconfigurable data flow from sensors to recognition results.

Macroprogramming model: (ATaG [69], Logical Neighborhoods [70], Kairos [71], Regiment [72]). Another approach for developing complex and large applications is *macroprogramming*, which considers a global behavior for a wireless sensor network, rather than single actions related to individual nodes. The need for this approach arises when developers have to deal with WSNs constituted by a large number of nodes, such that the complexity in coordinating their actions makes applications quite difficult to be designed in an effective way. Macroprogramming generally has some language constructs for abstracting embedded system's details, communication protocols, nodes collaboration, resource allocation. Moreover, it provides mechanisms through which sensors can be divided into logical groups on the basis of their locations, functionalities, or roles. Then, programming task decreases in complexity because programmers have only to specify what kind of collaborations exist between groups, whereas the underlaying execution environment is in charge of translating these high-level conceptual descriptions into actual node-level actions. Thanks to these high-level concepts, any domain expert not skilled in programming can develop its own application by simply defining the whole system behavior through concepts and terms they are familiar with.

Model-based approach: ([73]). It allows developers to define proper models representing the desired behavior of an application. Usually, such an approach consists in making use of a well-defined modeling language (finite state machine, flow charts, etc) and a tool capable of generating low-level code for a specific target platform starting from the model. Although it represents a standard methodology for several domains, such as automotive electronics, its employment in the WSN one has not been widely investigated yet. Then, very few tools have been actually proposed and tested.

Application-driven model: (MiLAN [74]). Middlewares belonging to this model aim to provide services to applications according to their needs and requirements, especially for QoS and reliability of the collected data. They allow programmers to directly access the communication protocol stack for adjusting network functions to support and satisfy requested requirements.

Virtual machine: (Maté [75], DAViM [76], DVM [77]). Virtual machines (VM) have been generally adopted for software emulating a guest system run-

ning on top of a real host. In the WSN context, VMs are used for allowing a broad range of applications to run on different platforms without worrying about the actual architecture characteristics. User applications are coded with a simple set of instructions that are interpreted by the VM execution environment. Unfortunately, this approach suffers from the performance overhead that the instructions interpretation introduces.

In Table 3.2, the previously described frameworks are summarized.

Table 3.2. Frameworks classification.

	Frameworks
Database model	TinyDB [56], Cougar [57], SINA [58]
Agent-based model	MAPS [59], AFME [60] Agilla [61], SensorWare [62], actorNet [63]
Domain-specific approach	SPINE [51, 64], Titan [65] RehabSPOT [66], BMF [67], CodeBlue [68]
Macroprogramming model	ATaG [69], Logical Neighborhoods [70] Kairos [71], Regiment [72]
Model-based approach	Hilac [73]
Application-driven model	MiLAN [74]
Virtual machine	Maté [75], DAViM [76], DVM [77]

On the basis of what we have reported, it emerges that none of the proposed application development methodologies can be considered the predominant one. Depending on specific tasks and/or contexts, a certain solution may result a better choice than others.

Most of them has peculiar features specifically conceived for particular application contexts but lacks in characteristics useful for more general-purpose uses. For instance, the frameworks based on a data centric approach provide high-level services just for data aggregation and querying but not for defining a more general-purpose computation. Thus, the data-centric model is not suitable for several domains requiring more sophisticated collaborative sensor data processing over the network.

For some specific context, e.g. the BSN-based wearable systems, most of these frameworks do not allow an explicit data flow processing over specific wearable nodes. This could be an important missing characteristic for many applications which are becoming crucial for the next future, such as context recognition, health monitoring, and medical assistance. In particular, gesture and activity recognition is a key operation for enabling a deep integration between persons and the spreading ubiquitous computing systems. In the BSN context, the physical locations of sensor nodes are very important and a well-defined framework should allow to compose a distributed data processing application by directly specifying a certain node, either because it is equipped with a particular sensor, or because it is close to the physical phenomenon

under observation, or simply because it just embodies specific computing capabilities. For instance, in an activity recognition application, it is important to know whether a specific sensing data flow comes from the sensor placed on the chest or on the leg, so to apply the proper processing task on it.

Further requirements for a software system providing high-level programming are fast application reconfiguration and platform independence. Reprogramming a network is a desirable features for supporting rapid and efficient changes of sensor nodes behavior. Systems like Deluge [78] and TinyCubus [79] provide code updates by directly loading them over the radio, but they require the use of a homogeneous platform (sw and hw) and also code transmission is a time and energy consuming operation. Virtual machines represent a typical approach for achieving a platform independent behavior. They allow to develop an application by using appropriate instructions that are interpreted by the VM running on sensor nodes. Unfortunately, this approach requires high computational and memory resources and can cause performance penalties because of the overhead in the code interpretation operation. Furthermore, writing an application with the provided instructions (e.g. Maté has more than a hundred instructions) is not fast and intuitive, especially if the application needs continuous changes.

3.2 SPINE2

In the light of what discussed in Section 3.1, the proposed SPINE2 framework has been specifically conceived as the ideal solution for enabling intensive distributed signal processing applications on such embedded systems. By providing a high-level visual programming language and a well-designed node-side architecture, SPINE2 is able to support heterogeneous sensor networks in a transparent way for the developer (it abstracts away any detail related to the underlying specific hardware) with negligible performance penalties. Moreover, it allows for an easy integration of new sensor drivers and processing functionalities as well as a fast porting procedure for supporting new sensor platforms. To the best of our knowledge, none of the research works proposed so far encloses all these characteristics in a unique programming framework for wearable sensor networks.

Intended to be used for a rapid development of effective yet efficient signal-processing applications, SPINE2 aims at providing developers an intuitive and straightforward design model based on simple graphical constructs. Specifically, a *task-oriented approach* has been chosen as high-level modeling paradigm for defining distributed applications on heterogeneous embedded environments. Also, since a WSN system usually demands very strict requirements in term of efficiency and stability, the SPINE2 middleware running on the nodes has been carefully designed and implemented so to guarantee high-performance execution of the task-based applications on the resource-constrained sensor embedded systems. Along with the efficiency aspect of the

node-side part of the framework, it is also very important to consider the reusability of the software components. To address such point, the underlying architecture has been also conceived to allow for an easy and fast porting procedure to support new sensor platforms.

In summary, SPINE2 has been designed to fulfill the following important requirements for a BSN programming framework:

- *High-level programming*: the use of programming methods based on high-level models can greatly improve productivity. By abstracting away any details related to the platform specific hardware or to the low-level communication protocol, a high-level programming layer is key for an effective and rapid application development.
- *Heterogeneity*: SPINE2 is able to deploy the same applications over different sensor platforms in a transparent way for the developer. This would allow the use of a unique development tool capable of managing heterogeneous sensor networks built up with different sensor node types.
- *Portability*: the node-side architecture has been designed to provide an easy and rapid portability toward new sensor platforms or any other C-like embedded system.
- *Extensibility*: the well-designed modular architecture allows for a very easy integration of new physical sensor drivers as well as new processing functionalities or communication capabilities, by minimizing time and efforts.
- *Efficiency*: the middleware running on the underlaying sensor platforms is lightweight and optimized so that the use of the framework in real world scenarios does not suffer from poor runtime performance.

3.2.1 Design and implementation

The SPINE2 framework has been designed as a powerful tool for a simple yet effective development of distributed signal processing application atop WSNs. In particular, its effectiveness derives from the adoption of a *task-oriented paradigm* providing simple constructs through which developers can specify the global behavior of the applications and easily manage application reconfiguration and reusability.

SPINE2 is composed of two main components: one runs on the coordinator of the WSN (tipically a PC or a smartphone) whereas the other is executed on the sensor nodes. The coordinator-side component, developed in Java, provides users with a very intuitive interface to the WSN. Specifically, it offers well-defined API through which the user-defined applications can easily manage the sensor network and quickly define, deploy, and run the task-oriented application. Moreover, it allows to gather the pre-processed data coming from the nodes and passes it to the user applications for more complex data processing and visualization. The node-side middleware, which runs on top of the sensor node operating system, has two main functions: (i) handling the messages coming from the coordinator and from other sensor nodes; (ii) interpretation and execution of the task specifications.

In the following, the main aspects characterizing the framework are described.

Platform independence and quick portability: SPINE2 has been designed for guaranteeing a rapid and simple portability process for supporting different sensor platforms. In particular, the node-side software architecture is conceived for decoupling the task runtime logic from services and features provided by the operating system of a specific platform. To this purpose, the *software layering* approach has been adopted (see Figure 3.1).

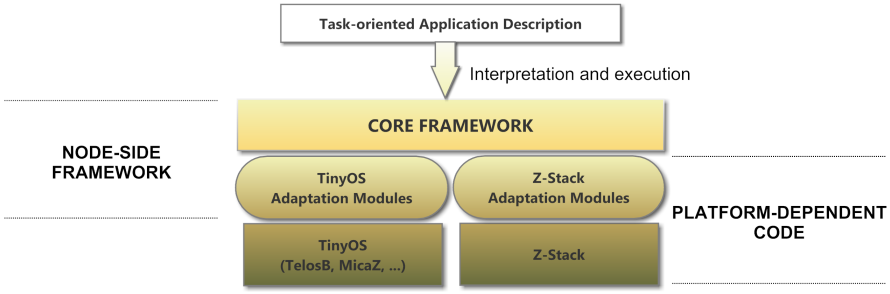


Fig. 3.1. The *Software Layering* approach for developing the framework.

According to such an approach, the node-side framework is designed so that a set of "core modules", developed in C and representing the main runtime system, constitutes the part of the software compatible with every C-like sensor platform. Along with these modules, other components constitute the platform-dependent part of the architecture and represent the adaptation interfaces between the core runtime system and the services and resources (sensors, timers, communication) provided by the operating system of a particular target sensor platform (such as TinyOS[4], and Z-Stack[5]). When doing a porting of the framework for a new sensor platform, the latter components are the only portion of software that a developer needs to implement.

Extensibility: the task-oriented design methodology allows to easily add new functionalities. This can be done by simply defining new tasks representing further computing capabilities without any need of changing the underlying runtime logic or the other task definitions. Moreover, supporting new hardware resources, such as sensors or actuators, can be also done straightforwardly.

Modularity: the framework architecture, composed of several and independent functional modules, allows for a more rapid development time and a more effective software maintenance and upgrading. For example, it may

[4] http://www.tinyos.net
[5] http://www.ti.com/tool/z-stack

be possible that future requirements need a different way for managing the memory or the task execution. Thanks to its modularity, the changes made by the developers affect only the correspondent modules without the risk of causing damages to the rest of the architecture.

3.2.2 The task-oriented approach

The SPINE2 task-oriented approach provides a set of programming abstractions for developing distributed signal processing applications by defining data-flow-oriented task chains. Such an approach is likely to be more intuitive and less error prone than explicitly writing code. In fact, an application can be simply specified as a set of interconnected tasks, which are chosen from the available task library, on the basis of the application requirements. Thus, the basic components constituting the high-level application model are *tasks* and *task-connections*. A task represents a well defined activity which can consist, for example, in a processing operation rather than a data transmission or a sensor reading. They are defined as atomic units of "work" that can not be subdivided. The atomicity of a task is only with respect to other tasks, as the event-reactive nature of the sensor nodes implies the need for a fast response to asynchronous events (a radio message reception or a timer expiration). A task-connection represents a relationship between tasks and generally has some kind of dependency, such as temporal and data dependency.

A typical sensor data processing application supported by the framework (see Figure 3.2) consists in (1) accomplishing the sensor readings, (2) passing the sensed data to processing functions which carry out some signal processing operations and (3) sending results to other nodes of the network (possibly for further elaboration).

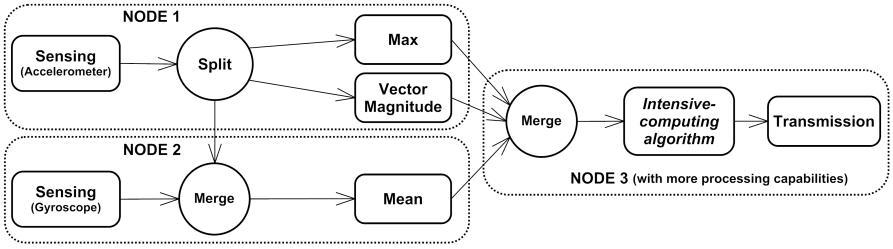
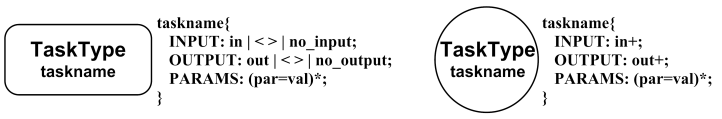


Fig. 3.2. Example of a task-oriented application with tasks instantiated on different nodes.

As the framework supports distributed data processing, developers can decide on which particular node each single task has to be performed, so that the execution of the whole application can be maintained well balanced. In

fact, depending on the different features of the nodes constituting the network, developers can allocate, for instance, the tasks requiring more resources to nodes providing more computational capabilities. The task-oriented representation, which captures both data and control flow, allows for a better application definition that will lead to effective scheduling activities and to a more efficient system implementation. Designing an application as a composition of basic blocks with fixed interfaces enables a rapid reconfiguration by the developer and then a more simple maintenance.

The defined library of *tasks* constituting the high-level application model are categorized in: *functional tasks* (Figure 3.3(a)) and *data-routing tasks* (Figure 3.3(b)). The former performs data processing/manipulation or execution control, whereas the latter provides data forwarding or replication.



(a) (b)

Fig. 3.3. Functional Task Description (a) and Data-Routing Task Description (b).

A task is defined through the following attributes: INPUT, OUTPUT, and PARAMETERS. In particular, the pair (*par=val*) represents a parameter setting of a task and, on the basis of the specific task (i.e. the TaskType), there may be zero or more parameters. The input and output attributes can have one of the following values:

- *"in"* or *"out"*: represents generic input and output. The user does not have to care about how data are formatted, and the actual data structure depends on the middleware that implements the language specifications. The symbol "+" in the data-routing description indicates the presence of multiple input/output connections.
- *"<>"*: indicates an *empty* input/output, i.e. it exists but does not contain any useful information. It can be intended like a simple "execution complete" notification that a task can issue to the next task.
- *"no_input"* or *"no_output"*: declares the absence of input/output. No tasks without both input and output are currently defined in the task library.

In the following, the tasks defined in the SPINE2 library are described.

- *TimingTask*: allows to define timers for timing other tasks. It does not have to elaborate any input data so it has no input link. Its function is to signal a notification in output when its inner timer expires. Task settings include the timer parameters: the periodicity (i.e. periodic timing or one-shot expiration), the period of expiration and the corresponding time scale/unit.

- *SensingTask*: performs a reading from a specific on-board sensor. Like the TimingTask, its configuration includes settings for the inner timer necessary for timing the sensing operation. It has no input whereas the provided output depends on the specific type of physical sensor it has associated. Since real sensors can provide more than a single sensed value, the output of a sensor can be seen as a set of "channels", each of which representing a sensed value (for instance, a triaxial accelerometer provides three different samples, i.e. acc-x, acc-y, and acc-z).
- *ProcessingTask*: performs data processing functions and algorithms, and constitutes the actual computing capabilities of the framework; typical functions are the so called "feature extractors" which are mathematical function applied to data series, such as Mean, Variance, etc.
- *TransmissionTask*: allows an explicit transmission of data coming from other tasks to a specific addressee. It is commonly used for sending pre-processed data or other information to the coordinator of the WSN. It is worth noting that, in case of connected tasks located on different nodes, the SPINE2 middleware will be in charge of the data transmission, without the need to explicitly make use of a TransmissionTask.
- *StoringTask* and *LoadingTask*: allow to store into or retrieve data from the flash memory placed on-board (if available) of the sensor node.
- *SplitTask*: is in charge of duplicating the data coming from its input to n outputs, so to make the same data available to a set of other tasks.
- *MergeTask*: merges data coming from its n inputs and provides the properly formatted collected data to a single output.
- *HistoricalMergeTask*: performs m merging operations over the time and maintain the collected data before making it available to the output.

3.2.3 The node-side software architecture

The node-side part of the SPINE2 framework represents the runtime system able to "interpret" and "execute" the high-level task-oriented application. Its architecture (shown in Figure 3.4) is well-structured and, by following a modularity approach, it comes with a set of independent but interacting modules, each of which has been intended to accomplish well defined operations.

The components in lighter color correspond to the *core framework* of SPINE2 (see also Figure 3.1). They are implemented in the C language, so that they can be easily compiled for practically every "C-like" compatible sensor platform, without the need for changing the code. Anyway, the very high portability of the framework does not only depend on the use of such a common language, but mainly on a strong software decoupling between the runtime execution logic (it encloses the unchangeable middleware layer logic, i.e. task and memory management, application-level message handling, abstract access to on-board sensors) and the components needed for accessing the specific services provided by the sensor platform on which the middleware is running. The grey blocks represent the *architeture-dependent* part of the

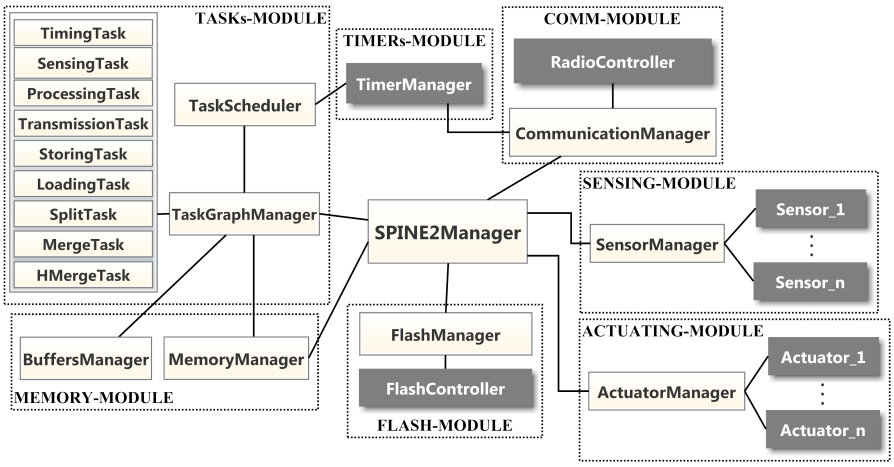


Fig. 3.4. Software architecture of the node-side part of the framework.

framework, i.e. software components needed to manage the node resources and thus tailored for a specific platform. They are adaptation components (or drivers) bridging the core with specific sensor platforms and guaranteeing the access to resources through well-defined interfaces.

The central architecture component is the *SPINE2Manager*. Its main functionalities are: (i) initializing the whole system at startup, (ii) regulating the access to the modules managing the node resources (radio, sensors, actuators, and flash memory) and (iii) handling the bidirectional high-level communication with both the user application on the coordinator and the other nodes, by means of the SPINE2 application-level protocol (see Section 3.2.5). On the basis of the incoming messages, the SPINE2Manager dispatches proper commands to the other components so to accomplish specific requested operations (e.g. a discovery phase or a task creation). It also takes care of the formatting of the SPINE2 outgoing messages, before delegating their encapsulation into low-level packets to the Comm-Module.

In the following, a description of the other modules depicted in Figure 3.4 is provided.

- *Comm-Module*: it manages the SPINE2 communication protocol by providing services for sending/receiving messages to/from the other sensor nodes and the WSN coordinator. In particular, it is in charge of encapsulating the SPINE2 application-level messages coming from the SPINE2Manager into well-formatted packets, as well as handling the reverse operation. It also provides (de)fragmentation operations, depending on the message length and on the maximum payload supported by the actual radio communication protocol adopted by a specific platform (see Section 3.2.5).

- *Tasks-Module*: it represents the "execution engine" of the SPINE2 middleware as it deals with the actual execution of the task-based user applications. In particular, it manages the set of SPINE2 tasks instantiated on the node, schedules them on the basis of their interaction relationship and properly supervises their execution.
- *Memory-Module*: it is responsible for managing the memory system both for supporting the SPINE2 internal mechanisms and for storing all the needed structures related to the user-defined application. Moreover, it is in charge of allocating the buffers used for data exchange among tasks or for supporting inner operations inside tasks (the user application may require a variable number of buffers each of which having an arbitrary size). Due to robustness motivations, most of the current operating systems for sensors do not allow dynamic memory allocation, so that a developer has to implement ad-hoc solutions. Such component has been conceived for supporting this capability. Thus, its main function is to provide other components a simple interface for allocating memory blocks on-demand at runtime.
- *Timers-Module*: it provides a common interface for dynamic allocation of timers, which are managed through a service based on the *publish/subscribe* paradigm. When an internal SPINE2 component (the subscriber) needs a timer, it makes a request to this module. If a timer resource is available, the subscriber is given its identification code through which it can be properly accessed for the subscriber's purposes. Similarly to sensors and actuators, timers depend on the hardware architecture: for each different sensor platform, proper timer drivers have to be provided in a well-defined interface so to be bound to the timer management module.
- *Sensing-Module*: designed for a simple management of the heterogeneous physical sensors placed on the sensor nodes by providing a standard interface for accessing them in a homogeneous and transparent way. Thus, each physical sensor driver has to be properly designed for being compliant to the Sensing module interface.
- *Actuating-Module*: similarly to the Sensing-Module, it guarantees an homogeneous use of the actuators installed on a sensor node, so that not only it is possible to sense from the environment but also to interact with it.
- *Flash-Module*: most of the microcontrollers that equip the common sensor platforms comes with a flash memory to permanently store information. This module provide the software logic required to access such a memory by means of a simple interface for data storing and loading.

At the moment, the node-side part of the framework has been implemented and tested on sensor platforms running TinyOS and on a custom mote based on Z-Stack, which is the ZigBee-compliant implementation provided by Texas Instruments.

3.2.4 The coordinator-side architecture

The *SPINE2 Coordinator* represents the access point to the WSN for a user/developer by providing a well-defined software interface. In particular, in order to manage a SPINE2-based network, users can develop their own applications on top of the SPINE2 Coordinator by easily exploiting the offered simple and intuitive API. Such an API allows for (i) controlling the remote nodes, (ii) defining, deploying, and controlling the task-oriented application, and (iii) gathering pre-processed data coming from the sensor nodes. When the application is "registered" with the SPINE2 Coordinator, it is enabled to get notified by the high-level events generated by the WSN: discovery of new nodes, errors coming from the network, new sensor data transmission, etc. Such a user-defined application is usually in charge of performing further computation on the pre-processed sensor data coming from the WSN.

To enhance portability, Java has been adopted for implementing the coordinator-side architecture of SPINE2. Moreover, it is worth noting that, since none of the existing computer or mobile devices have a native wireless interface for communicating with the 802.15.4-based radio equipping the common sensor platforms, a portion of the implementation is strictly dependent on the specific base-station module (providing proper radio communication capabilities) connected to the coordinator.

A simplified Package Diagram of the SPINE2 Coordinator is depicted in Figure 3.5.

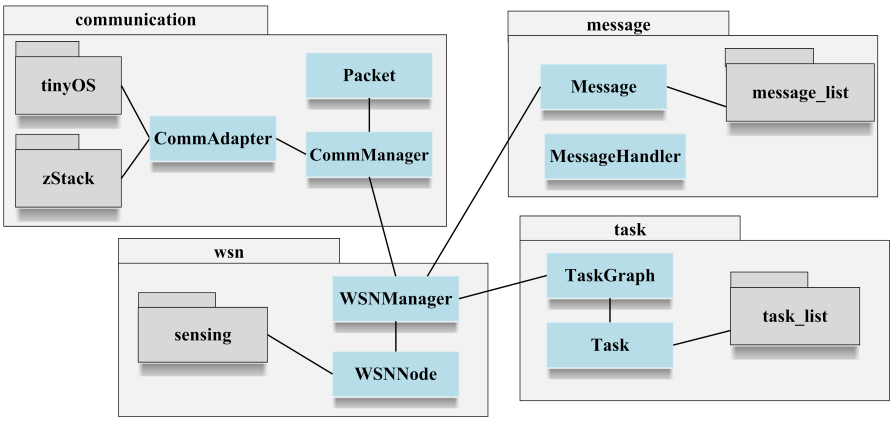


Fig. 3.5. Software architecture of the SPINE2 Coordinator.

The *Communication Package* includes all classes needed for interacting with the coordinators and the sensor nodes of the WSN. In particular the *CommManager* is the class responsible for managing the lower level of the

two-layer stack communication protocol supported by the framework (see Section 3.2.5). It provides the service for encapsulating application level messages into the SPINE2 Packet defined in Section 3.2.5 (the *Packet* class represents this data transmission unit). To this purpose, it is in charge of performing the necessary fragmentation if the message length exceeds the maximum payload supported by the low-level protocol of a specific sensor platform. The CommManager does not have to care about the actual way to communicate with a specific platform as it can rely on a set of packages (one for each supported platform), each of which encloses specific platform-dependent class definitions. These classes are uniformly accessed through a unique and standard interface, i.e. the *CommAdapter*.

The *WSN Package* is responsible for managing information about the network. In particular, the *WSNManager* contains an internal representation of the nodes constituting the network whereas node characteristics, such as platform, physical address, and the list of supported on-board sensors and tasks are described in the *WSNNode* class. The WSNManager is also responsible for managing the application-level communication protocol, i.e. the upper layer of the two-layer stack communication protocol supported by the framework. Hence, it handles the SPINE2 messages (see Section 3.2.5). The WSNManager allows deploying the user defined task-based application over the network. By interacting with the *TaskGraph*, it gets information about tasks and connections to be instantiated on each node. Then, after having created proper messages (i.e. the CreateTask Message and CreateConnections Message, see Section 3.2.5), it sends them by means of the transmission service provided by the CommManager.

The *Task Package* includes class definitions used to maintain a consistent representation of the user defined task-oriented application. In particular, the *TaskGraph* actually encapsulates the task-graph description of the final application. The *task_list package* represents the library of tasks currently supported by the framework, all based on a common representation, the *Task* component.

Finally, the *Message Package* encloses the list of all the defined application-level messages as well as the *MessageHandler*, which is in charge of handling the messages coming from the sensor nodes.

To better support the developers, the *SPINE2 Console* (see Figure 3.6) is also provided. It is a ready-to-use and user-friendly tool through which it is possible to exploit the SPINE2 functionalities without an explicit use of its Java API: the task-based application can be quickly modeled and deployed through an intuitive graphical interface and information about the nodes can be easily displayed. In particular, the GUI allows developers to easily manage the following operations: (1) discovery of the nodes constituting the WSN, along with their capabilities in terms of supported tasks and physical sensors; (2) management of the task-graph application, i.e. definition (including task parameter setting), deployment, file saving/loading; (3) log of all events com-

ing from the WSN including the data pre-processing results of the application and error messages.

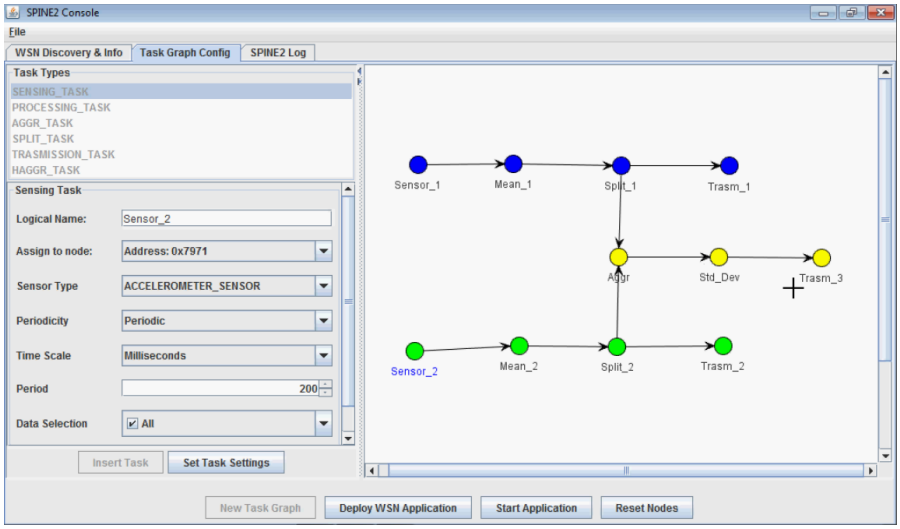


Fig. 3.6. SPINE2 Console.

3.2.5 SPINE2 communication protocol

The interaction between the SPINE2 Coordinator and the sensor nodes relies on a two-layer communication stack built atop the actual platform-dependent low-level communication protocol provided by the on-board radio of the sensor devices (see Figure 3.7(a)).

The lowest platform-independent layer provides a simple point-to-point communication. It notably handles the fragmentation of the high-level messages into multiple packets, so that the maximum payload length supported by the radio protocol is not a limitation for the amount of data to be transmitted. The packet format handled on this layer is shown in Figure 3.7(b).

The highest communication layer manages a set of pre-defined application-level messages, which encapsulate the commands and information for defining and interacting with the WSN and, more specifically, with the task-oriented application built and deployed by the user. In particular, the SPINE2Manager depicted in Figure 3.4 is the component handling these messages on the node-side part of the framework.

The defined application-level messages are summarized in Table 3.3 along with their source-destination direction and, if any, further information and parameter carried in the message payload.

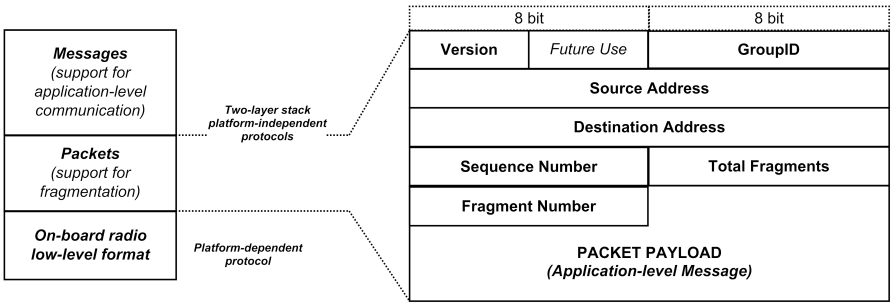


Fig. 3.7. The framework protocol stack layers (a) and Packet definition (b).

Table 3.3. List of SPINE2 messages.

Message Type	*Source*	*Destination*	*Payload*
Discovery Nodes	Coordinator	Node	-
Create Task	Coordinator	Node	task configuration
Create Connections	Coordinator	Node	connection configuration
Init Application	Coordinator	Node	-
Start Application	Coordinator	Node	-
Reset Application	Coordinator	Node	-
Node Advertisement	Node	Coordinator	node info,sensors list,tasks list
Node Application Ready	Node	Coordinator	-
Sensor Data	Node	Coordinator	formatted data
Error	Node	Coordinator	error code, error info
Status Info	Node	Coordinator	status code, status info
Sensor to Sensor Data	Node	Node	formatted data

Most of the messages are self-explanatory, like the ones adopted for controlling the execution of the deployed task-based application, i.e. the *Init Application*, *Start Application* and *Reset Application Message*, which do carry any information as payload.

The *Discovery Nodes Message* is usually the first message for initiating the communication scheme between the coordinator and the sensor nodes and is issued for requesting general information about the nodes (e.g. their sensor platform type) along with the list of physical onboard sensors and the list of task types that they are actually capable to instantiate. Such an information is sent by each node of the WSN through the *Node Advertisement Message*.

The Discovery/Advertisement phase takes place before the application deployment, so that each task can be associated to a node belonging to the current WSN. After this phase and after the user has finished modeling its application, the deployment phase can start and consists in mapping the task-graph over the network. The *Create Task Message* is sent by the coordinator for instantiating a task, along with its parameters, to a specific node. Similarly, the *Create Connections Message* is sent for creating a connection, or a

set of connections, among tasks. Thus, such a message also encloses information related to the destination task of a specific connection; it is either local (i.e. instantiated on the same node) or remote. Furthermore, it includes information for configuring the buffers to be allocated on the node corresponding to the instantiated connection.

Once the application has been deployed, the coordinator broadcasts the *Init Application Message* for initializing all task instances over the network. Once the initialization has been correctly accomplished, each node replies with a *Node Application Ready Message*, indicating that the application is ready to start. A *Start Application Message* can be then broadcast so causing the application to run.

The *Sensor Data Message* is used by the node when properly formatted sensor data (raw or pre-processed) have to be sent to the coordinator. The *Sensor to Sensor Data Message* is, instead, necessary when interconnected tasks are distributed on different nodes: it encapsulates data produced by a task that needs to be sent to a remote destination task.

Finally, the *Error* and *Status Info* messages can be issued by a node in case of unexpected errors during initialization or application runtime (e.g. when it is not possible to allocate further blocks in the dynamic memory) or for periodic node status advertisement (e.g. remaining node battery level).

3.2.6 Developing applications

After having described the main elements constituting the SPINE2 framework (i.e. the node-side and coordinator-side components, and the communication protocol), in Figure 3.8, a high-level view of the whole SPINE2 environment is shown, along with its relationship with the Java-based application defined by the user.

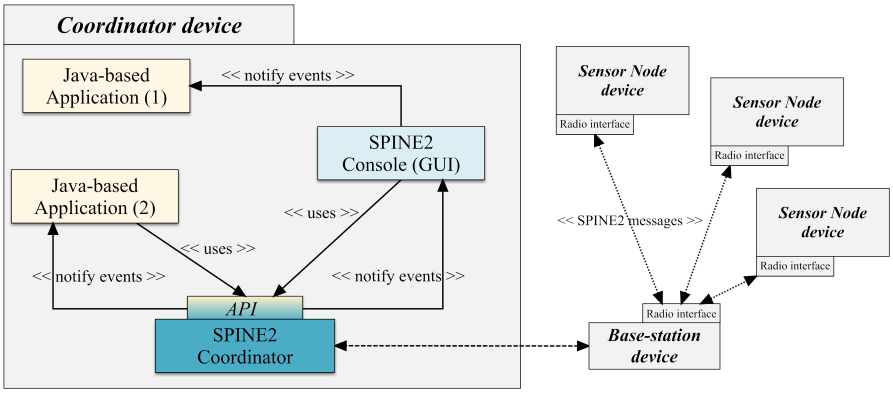


Fig. 3.8. Interaction between user applications and the SPINE2 components.

As it can be seen, the user applications can be interfaced to the SPINE2 framework in two different ways. In a first example (*Application 1*), it is supposed that the developer makes use of the SPINE2 Console for managing both the WSN and the task-based application. In this case, the application just needs to be registered to the Console and does not have to be implemented any extra code rather than the one related to possible further processing/-management/display functionalities on the data coming from the nodes (new available sensor data is directly notified by the Console). On the contrary, in the second example (*Application 2*), the application directly interacts with the SPINE2 API and then needs to take control of all the aspects related to both node management and task-based application development by means of Java code. As a consequence, it requires a little more efforts and time from the developer.

In Figure 3.9, the typical phases involved in the development and deployment of a task-based application on the sensor nodes are shown. In particular, the messages exchanged between the SPINE2 Coordinator and a generic node, as well as the interactions among some of the inner modules composing the node-side architecture are depicted. For the sake of clarity, in the diagram, the developer directly interacts with the SPINE2 Coordinator but, actually, either the SPINE2 Console or a user-defined application stands in between. Anyway, in both cases, the commands issued to the SPINE2 Coordinator represent calls to the SPINE2 API.

The communication scheme between the Coordinator and the network is usually initiated with a node discovery phase, in which the Coordinator broadcasts a discovery request (by means of the Discovery message) and waits for the advertisements from the active nodes within range. Such a request, which can also be sent on a periodical basis to keep the network information updated, is handled by the SPINE2Manager, which replies with a Node_Advertisement message. This message contains all the necessary information about the node, such as, among the others, its sensing capabilities and the type of tasks supported. Once the developer (or his Java-based application) has a complete knowledge of the network, he/she can start building the distributed task-based application by means of a series of API calls for creating and setting the desired tasks and their interconnections. The application definition phase only involves the SPINE2 Coordinator, and the application model is locally managed and stored on the coordinator device until the developer issues a Deploy command. As a consequence, the SPINE2 Coordinator first performs a final check to be sure that the application model has been correctly defined, i.e. all task are correctly connected, configured and assigned to a specific sensor node. If the correctness check fails, a local warning is triggered and the deployment phase cannot go further. Otherwise, the Coordinator generates a series of Create_Task and Create_Connection messages, which are sent to the proper nodes. On each node, the SPINE2Manager takes care of these requests and, consequently, interacts with the TaskGraphManager for instantiating tasks

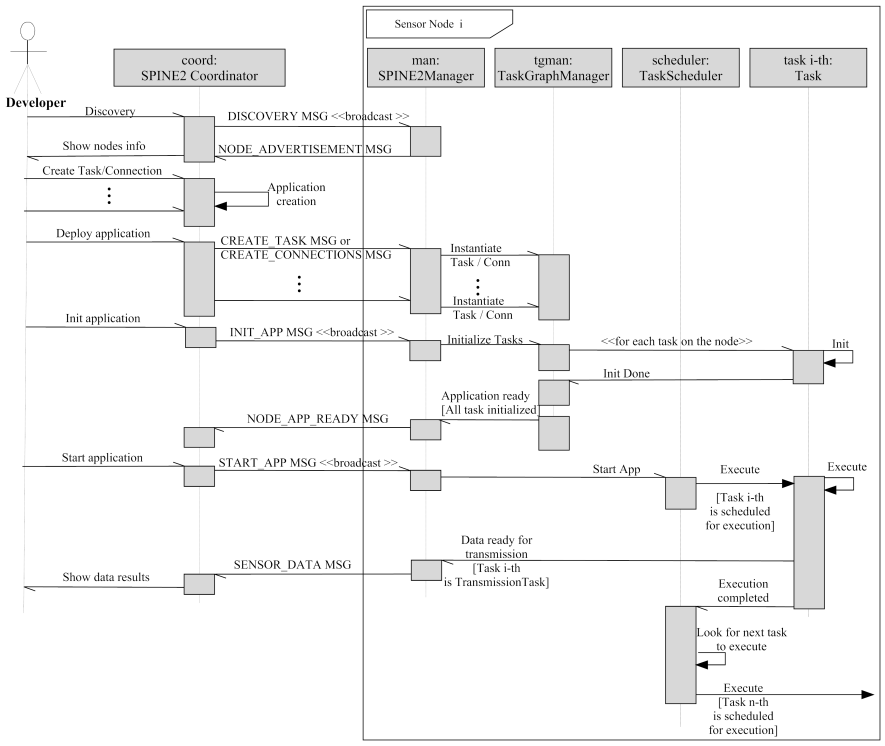


Fig. 3.9. Sequence diagram for the SPINE2 application development process.

and connections, which are stored in a properly defined data structure. Once the deployment phase is over, the distributed task-based application needs to be initialized by means of a Init_App message, which is broadcast over the network. The TaskGraphManager makes sure that all the task instances on each specific node are initialized. The initialization phase is important because some tasks may have the need to use specific data structures, which have to be correctly instantiated. Once all tasks have been correctly initialized, the SPINE2Manager can send a Node_App_Ready message to the Coordinator, notifying that the node is ready for the execution of the task-based application. Anyway, the developer is allowed to run the remote application only when all nodes in the network have successfully sent such a message. Upon the reception of a Start_App message on each node, the application is eventually started, and the SPINE2Manager simply delegates the TaskScheduler for the actual execution of the task-based application. On the basis of the structure of the application instantiated on the node, the TaskScheduler selects the first task to be executed and waits for its termination. When its execution is completed, the scheduler chooses the next one and the process goes on as

long as there is at least a task to be executed. It is worth reminding that, although the sequence diagram of Figure 3.9 depicts the mechanisms between the Coordinator and a single generic node, SPINE2 is able to manage the execution of interconnected tasks residing on different sensor nodes. This is done by transmitting the data produced by the source task to a remote node, on which the TaskScheduler is in charge of passing the received data to the specified destination task and, consequently, triggering its execution. In particular, the SPINE2Manager on the source node is in charge of arranging data in a well-defined format and encapsulating it into a Sensor_to_Sensor_Data message (not shown in the diagram). Similarly, when data produced by a task needs to be transmitted to the Coordinator (only the TransmissionTask is currently used for that purpose), the SPINE2Manager encapsulates it into a Sensor_Data message.

To demonstrate how effortlessly a developer can make use of the SPINE2 API, in Listing 3.1, a Java application example is shown. In particular, its purpose is to get information from a sensor network and define a simple task-based application, which is deployed on a single node. It is worth noting that, for the sake of clarity and simplicity, the example is provided as simple as possible, without considering complex logics or the handling of possible exceptions.

The central class of the SPINE2 Java library is the SPINE2Manager, which represents the main entity interfacing the user-defined application with the WSN. In order to get notified of the SPINE2 events, the application needs to (i) implement the SPINE2Listener interface and (ii) register itself as one of the listener of the SPINE2Manager, whose instance is first generated by a "factory-class" on the basis of the specific properties set in the file passed as parameter (see lines 6-7). Afterwards, it is possible to issue a node discovery phase (line 8). As a consequence, as soon as a node replies, the application is notified, i.e. the *nodeDiscovered* method is called. Thus, the developer is able to get all the information about that node, such as its platforms, its address, the list of sensors it is equipped with, and the list of available tasks it is possible to instantiate on it (see lines 10-17). Moreover, after a certain timeout period, the SPINE2Manager also notifies the application of the end of the discovery phase by calling the *discoveryCompleted* method (lines 18-24). In this specific example, the nodes are just stored in a local data structure for future use and the *createApp* method is called. The logic inside this method consists in defining a simple task-based application constituted of three tasks (Sensing, Processing and Transmission) to be instantiated on the first of the nodes that have replied during the discovery phase. Each task is first instantiated and configured with proper settings and then added to a TaskGraph instance, which is in charge of storing the whole application structure. Specifically, the SensingTask is bound to the accelerometer sensor, which is read on all the axes (see line 34), periodically every 50ms. Moreover, the task is set to notify its following task (the processing one) after having collected 10

samples (see the "outputBuffering" parameter in line 35). The ProcessingTask performs the mean function every 10 samples (the *shift* parameter) on a window readings of 20 samples and forward the results to the next task after having collected 5 mean values (line 43). Finally, the TransmissionTask is set to transmit the received 5 mean values to the base station device (i.e. the SPINE2 Coordinator). After having instantiated, configured and added the tasks to the taskgraph, the connections are eventually created by specifying both the source and the destination tasks (lines 50-51).

The code displayed in lines 54-56, consists in defining methods performing calls to other important SPINE2 API functions, i.e. the ones for deploying, starting, and resetting the previously built task-based application. Specifically, the second parameter of the *deployApplication* specifies if the application has to be automatically started as soon as the deployment phase is finished (i.e. every nodes replies with a Node_App_Ready message to the Coordinator) or if it is required an explicit call to *startApplication*. The latter one is the case of the example, since the parameter is set to "MANUALLY_START_APPLICATION".

Finally, the method *messageReceived(Message msg)* (lines 58-62), is automatically called by the SPINE2Manager when a new message from a node is received by the Coordinator. Since there exist different types of SPINE2 messages (see Section 3.2.5), the developer can distinguish them by simply check which class the specific message instance belongs to. Moreover, further SPINE2 API functions can be used to get and manage all the information encapsulated into a certain message, such as the sensor data coming with a "SensorDataMessage".

3.2.7 Performance evaluation

In this section an evaluation of the framework is provided for motivating its use as an effective and easy supporting tool for rapid development of WSN applications. SPINE2 has been developed for providing a lightweight application execution engine. Thanks to an accurate integration of the *task-oriented* application modeling and the *software layering* architecture, it is ensured that no significant overhead can lead to performance penalties.

In the following, specific measures related to processing and memory performances of the node-side part of the framework are carried out to demonstrate how SPINE2 is suitable for embedded resource-constrained environments such as the sensor nodes. Specifically, the results are obtained both for TelosB motes running TinyOS and for a custom platform based on Z-Stack. A summary of the main characteristics of these two sensor platforms are shown in Table 3.4.

3.2.7.1 Computational performance

As SPINE2 has been mainly designed for the development of distributed signal processing applications on embedded systems like the WSNs, the core of

Listing 3.1. Example of use of the SPINE2 API

```
2  public class SPINE2Test implements SPINE2Listener{
3     private SPINE2Manager manager;  private TaskGraph tg; WSNNode motes[]= null;

5     public SPINE2Test(){
6        manager = SPINE2Factory.createSPINE2Manager("file.properties");
7        manager.addListener(this);
8        manager.discoveryWSN();
9     }
10    public void nodeDiscovered(WSNNode node){
11       //get some info from the node and...
12       String address = node.getAddress();
13       String platform = node.getPlatform();
14       Vector sensorsList = node.getSensors();
15       vector tasksList = node.getTasks();
16       // ... somehow display its characteristics
17    }
18    public void discoveryCompleted(LinkedList nodes){
19       motes= new WSNNode[nodes.size()];
20       Iterator it = nodes.iterator(); int i=0;
21       while(it.hasNext())
22          motes[i++]= (WSNNode)it.next();
23        createApp();
24    }
25    private void createApp(){
26       tg= new TaskGraph();
27    // SENSING TASK
28          SensingTask st= new SensingTask(motes[0]);
29          st.setLogicalName("Sensing");
30          st.setPeriodicity(SensingTask.TIMER_PERIODIC);
31          st.setPeriod(50);
32          st.setTimeScale(SensingTask.TS_MILLISEC);
33          st.setSensorType(Sensor.ACCELEROMETER);
34          st.setDataSelection(SensingTask.DATA_ALL);
35          st.setOutputBuffering(10);
36          tg.addTask(st);
37    // PROCESSING TASK
38          ProcessingTask ptMean= new ProcessingTask(motes[0]);
39          ptMean.setLogicalName("P_Mean");
40          ptMean.setFunctionType(FunctionConstants.F_MEAN);
41          ptMean.setWindowSize(20);
42          ptMean.setShiftSize(10);
43          ptMean.setOutputBuffering(5);
44          tg.addTask(ptMean);
45    // TRANSMISSION TASK
46          TransmissionTask transmT= new TransmissionTask(motes[0]);
47          transmT.setLogicalName("Transmission");
48          transmT.setDestinationAddr(CommConstants.SPINE_BASE_STATION_ADDR);
49          tg.addTask(transmT);
50       tg.addConnection(st, ptMean);
51       tg.addConnection(ptMean, transmT);
52    }

54    void deployApplication(){manager.deployApplication(tg,WSN.MANUALLY_START_APPLICATION);}
55    void startApplication(){ manager.startApplication(); }
56    void resetApplication(){ manager.resetApplication(); }

58    public void messageReceived(Message msg){
59       if(msg instanceof SensorDataMessage){
60          //manage the sensor data encapsulated in the message
61       }
62    }
63    public static void main(String[] args){ new SPINETest(); }
64 }
```

Table 3.4. Main characteristics of TelosB and Z-Stack-based sensor platforms.

	CPU	RAM (KB)	Program Memory (KB)	OS
TelosB	TI MSP430 - 8MHz, 16bit	10	48	TinyOS
Z-Stack platform	CC2430 - 32MHz, 8bit	8	128	Z-Stack

the functionalities provided to developers is constituted by a set of processing functions. They are all made available through a unique abstraction, the ProcessingTask, which can be configured on the basis of the actual needs.

Specific functions, called *feature extractors*, are commonly used for executing in-node aggregated math computation (e.g., max, min, standard deviation) on sensed data before transmitting them to a more powerful device. Since many of the WSN applications include real-time operations, such as activity recognition or human body monitoring, they require an efficient execution performance to be successfully employed for managing data coming from sensors with high sampling rates. To satisfy this requirement, the task execution model of SPINE2 has been designed and implemented to be lightweight so guaranteeing high efficiency.

In Table 3.5, the average execution times of different types of ProcessingTask over different data windows are shown. Such tests have been executed on a simple application deployed on a node that involves a SensingTask connected to the ProcessingTask. In particular, processing involves the *min*, *mean*, *standard deviation*, *vector magnitude* and *pitch and roll* functions computed on a triaxial accelerometer data stream.

Table 3.5. ProcessingTask execution time over a 3-channel sensor data [ms].

	TelosB					Z-Stack custom platform				
window size:	*10*	*50*	*100*	*150*	*200*	*10*	*50*	*100*	*150*	*200*
Min	0.67	1.06	1.55	2.04	2.53	0.37	0.57	0.81	1.04	1.25
Mean	0.94	1.68	2.17	2.78	3.30	0.52	0.91	1.14	1.42	1.61
Standard Deviation	3.99	8.19	15.68	21.78	27.53	2.12	4.43	8.2	11.12	13.63
Vector Magnitude	2.09	4.64	8.18	12.33	15.81	1.16	2.51	4.28	6.29	7.83
Pitch and Roll	17.81	18.14	18.70	19.31	19.82	9.80	9.88	9.97	10.04	10.11

3.2.7.2 Memory usage

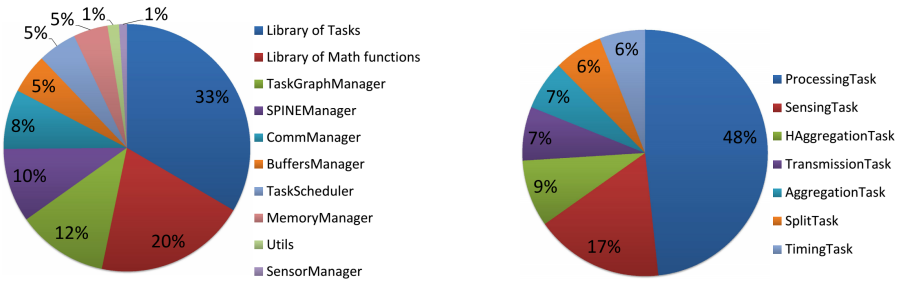
In this section an evaluation of the program and RAM memory usage is provided. In particular, Table 3.6 shows the memory amount related to both the "core framework" and the platform-dependent code. The latter includes the SPINE2 adaptation modules adapting the "core" to the specific sensor architecture as well as the sensor drivers equipping both platforms.

Table 3.6. ROM (Program) and RAM Memory usage [Bytes].

		TelosB-TinyOS (ROM/RAM)	*Z-Stack (ROM/RAM)*
SPINE2 Core framework	Managing components	9875 / 3403	
	Library of tasks	4956 / 566	
SPINE2 platform-specific code		2567 / 196	2891 / 107
Operating System		24572 / 1879	69681 / 3465
Total		*41970 / 6044*	*87403 / 7541*

It is worth noting that the RAM usage related to the core framework can be variable as developers can easily configure SPINE2 at compile-time with the proper amount of dynamic memory (they can specify both the buffer space used for data exchange among tasks and the memory space used for storing the task-based application). Specifically, the data shown in Table 3.6 includes 800-bytes space of memory used for storing the task-based application and 1600-bytes space for dynamic buffer allocation. Such an amount of memory is usually sufficient for instantiating quite complex user applications including tens of tasks on each single sensor node. In fact, the RAM needed for allocating an instance of task is relatively small, ranging from 13 bytes (SplitTask, MergeTask, and HystoricalMergeTask) to 28 bytes (ProcessingTask); 21 bytes are required for a SensingTask, whereas a single Connection needs 13 bytes.

For a more detailed view on how the program memory is used by SPINE2, the chart in Figure 3.10(a) shows how the ROM usage is distributed among the main functional components of the SPINE2 core. In particular, the set of tasks and the library of mathematical functions represent almost half of the entire required memory. The remaining memory usage is distributed among the modules devoted to managing different architectural aspects (e.g. task-execution, memory, timers, communication, sensors).



(a) (b)

Fig. 3.10. ROM usage by SPINE2 core components (a) and tasks (b).

Moreover, as shown in Figure 3.10(b), the ProcessingTask is the one that requires the most amount of memory. This is because it has to implement the

logic devoted to executing the various signal-processing functions, including managing input stream and processing results. All other tasks require a lesser amount of memory, since they implement a lesser complex logic. Table 3.6 provides an indirect information about the amount of code of the platform-independent and -dependent parts of SPINE2 under the memory usage point of view. However, to better understand the actual benefits offered by the SPINE2 architecture, the percentage of code lines can be considered. In particular, for both sensor platforms, only roughly one-fourth of the total code of SPINE2 is represented by the platform-specific one (specifically, 23% for TinyOS and 26% for Z-Stack). It is worth noting that this comparison takes into account the currently supported sensors (accelerometer, gyroscope, temperature, voltage) and not all possible physical sensor drivers. Nevertheless, since all "core functionalities" (e.g. signal processing functions and data-flow management) are already defined, the effort developers need to put for a complete framework porting is much lesser than if they have to deal with a software architecture designed for a specific sensor platform.

3.2.8 Related work and comparison

Among the programming frameworks for WSN/BSN, Titan [65] and ATaG [69], similarly to SPINE2, provide a task-oriented approach.

Titan [65] is a framework specifically designed to support context recognition on WSNs. A processing application is defined by a set of interconnected tasks which are executed by the runtime as a whole over the network. Each single task is actually mapped and executed by a specific sensor node, according to its resources, and represents a specific operation: an algorithm for data processing like a classifier, a mathematical function or an operation to access the sensor node hardware. Titan requires the presence of a node which has to guarantee enough resources in terms of performance execution and memory. This is because such a node is in charge of managing the distributed application over the sensor network. First of all, it discovers the actual capabilities of sensors and the list of their available tasks and then, on the basis of the entire task network defined by the user, it determines which subset of tasks has to be assigned to each node. Titan has been implemented atop TinyOS and has been tested only on the Tmote Sky sensor platform.

The Abstract Task Graph (ATaG) [69, 80] represents both a method and a programming language for developing WSN applications through a macro-programming approach. This means that developers have to specify a global behavior for the application and the framework translates it into node-level specifications. An ATaG application is defined as a set of "abstract tasks" representing computational operations and a set of "abstract data items" to be exchanged between abstract tasks. Every task has well-defined input/output interfaces (named "channels") which declare the data items the task consumes in input and the ones it produces as output. The "abstract" adjective indicates that the number and the placement of the user-defined tasks

are determined at compile-time depending on the actual target network. The task runtime abstraction consists in a sort of data-driven control flow mechanism, since tasks do not directly interact with each other but, on the contrary, they interact with the only data items, which are stored in a data pool, through their input/output channels. Thus, a task is scheduled for execution only when appropriate input data is available. The framework provides a hybrid imperative-declarative programming style for defining user applications, which allows to separate definitions of "when" tasks fire and "where" tasks have to be deployed on a sensor network from "what" each single task has to do when in execution. The former is provided by a high-level declarative approach, while the latter is provided by imperative code implementing the actual processing operation on input data. The declarative part of an ATaG program consists in a graphical approach providing a set of abstract elements (i.e. tasks, data items and channels) which have to be properly defined and interconnected. Starting from the above tasks description, the ATaG compiler produces a skeleton code which needs to be completed with low-level imperative code implementing the actual processing job for each defined abstract task. This code completion is up to the developer. ATaG currently supports only the Java-based Sun SPOT[6] sensors.

SPINE [51, 64] is an open-source, programming frameworks specifically tailored for developing signal-processing applications on BSNs. While providing an effective programming by means of high-level abstractions, it guarantees high efficiency thanks to its architecture specifically designed for operating in the BSN domain. The BSN architecture supported by SPINE is a star topology constituted by one coordinator and multiple sensor nodes. The support to a wide set of pre-defined physical sensors and signal-processing utilities ensures rapid prototyping by minimizing design time and efforts. Its libraries include most of the common signal processing algorithms used in BSNs; for instance, SPINE supports distributed classification where feature extraction functions are computed on the sensor nodes and the results are sent to the BSN coordinator. SPINE currently supports the most diffused sensor platforms based on TinyOS such as TelosB/Tmote Sky, MicaZ, and Shimmer I and II, as well as the Z-Stack and the Java-based Sun SPOT. The sensor-node side of the framework includes several utilities for signal processing such as data storage buffers, mathematical function libraries and common feature extractors used in signal processing. The coordinator component (running on PCs or smartphones) consists of a Java-based interface in charge of enabling the communication between the user PC and the sensor nodes through a specific SPINE protocol. It also acts as a gateway for remote data access and provides proper APIs for managing and setting the sensor nodes and issuing service requests. On top of the coordinator-side component, developers can build their own applications for collecting sensor data and making further analysis on it. The modular architecture of SPINE allows for an easy inte-

[6] http://www.sunspotworld.com

gration of new custom-designed sensor drivers as well as processing functions so to guarantee a straightforward easy framework extensibility over time to meet the particular needs of specific applications.

HILAC [73] is a model-based framework for modeling, simulation and automatic code generation for multiple sensor platforms. It is built on top of the Math-Works tool chain and allows developers to model applications using Stateflow/Simulink[7] having no detailed knowledge of the target hw/sw platform. Once the application model has been defined, behavioral simulation and functional verification can be performed, as well as HIL (Hardware-In-the-Loop) simulation by interacting with real sensor nodes. The framework is also able to generate ANSI C code from the Stateflow representation, whereas proper Target Language Compiler (TLC) scripts extract sections of the code and add platform-specific source code for different sensor platforms. HILAC currently supports TinyOS-based platforms, MANTIS-based nodes, and the ZigBee stack provided by Ember.

In Table 3.7, a comparison between SPINE2 and the aforementioned frameworks with respect to several characteristics is provided.

Table 3.7. Comparison between SPINE2 and other programming frameworks.

	SPINE2	SPINE	Titan	ATaG	HILAC
WBSN-specific framework	X	X	X		
Specific support to in-node signal processing	X	X	X		
Task-oriented abstraction	X		X	X	
Multi-platform support	X	X			X
Common platform-independent architecture	X				
Platform interoperability	X	X			
Easy extensibility	X	X			
Quick porting	X				
Application reconfiguration at runtime	X	X	X	X	

SPINE2 shares most of the listed features with SPINE, although the proposed framework inherits from its predecessor just its philosophy-of-use and the purposes. In fact, the programming abstractions are completely different whereas the node-side architecture has been deeply reengineered both for managing the task-based approach and for embracing a platform-independent modeling. The notion of "core framework" is not present in SPINE in which the multi-platform support can be enabled only by means of specific portings made available for different sensor platforms.

Titan has been conceived for developing BSN applications, by specifically supporting signal processing functions on the nodes. It also exploits a similar

[7] http://www.mathworks.com

high-level task-oriented paradigm but, since no GUI is provided and the developers need to use a text file for configuring the network of tasks, complex applications composed of many tasks are hard to define and manage. Differently from SPINE2, its node-side runtime architecture is not designed for platforms different from the TinyOS ones, and also a porting process seems to be quite difficult since TinyOS-specific constructs are deeply coupled to its features and services.

Also ATaG makes use of a task-oriented approach for WSN applications. In particular, in the offered approach, an explicit way for defining data items used in the applications is adopted. On the contrary, in SPINE2 this is not necessary because the presence of data is implicit into the definition and implementation of the tasks. Thus, the user does not have to worry about it as data is implicitly managed by composing the workflow. Moreover, the ATaG system demands the actual implementation of each defined task to the users, by means of a low-level programming language. Specifically, ATaG is currently available only for Sun SPOTs.

The HILAC programming framework has not been specifically conceived for BSN applications development, but it is capable of supporting different sensor platforms. Unfortunately, the major drawback is that its code-generator approach produces source code that needs to be manually uploaded on each physical node (unless an OTA programming functionality is available). For this reason, reconfiguring an application at runtime is not possible.

In the following, some performance evaluations on SPINE2, SPINE and Titan are also presented. In particular, measurements of the memory usage of the frameworks, energy consumption and bandwidth under a given application profile is provided. The benchmark consists in reading from a triaxial accelerometer at a sampling rate of 20Hz and transmitting a sequence of 20 sensor samples to the coordinator without any in-node pre-processing (i.e. a data message is generated every second). Results are shown in Table 3.8.

Table 3.8. Performance comparison between SPINE2, SPINE, and Titan.

	Memory req. (used/av.)		*Energy consumption*		*Bandwidth*
	RAM [KB]	*ROM* [KB]	*Avg. power* [mW/s]	*Lifetime* [h]	[bytes/s]
SPINE2 on TelosB	5.9 / 10	41 / 48	7.5	87	166
SPINE on TelosB	3.7 / 10	33.5 / 48	6.6	101	160
Titan on TelosB	9.0 / 10	38.7 / 48	18.7	36	158
SPINE2 on Z-Stack	7.3 / 8	85.3 / 128	12.7	51	182
SPINE on Z-Stack	3.9 / 8	95.9 / 128	11.2	60	176

The memory required by each framework has been analyzed by considering both the framework itself and the underlying operating system, as well as the drivers of a custom sensor board equipped with the accelerometer. The amount of ROM needed by SPINE2 on the TinyOS platform is greater than

the one required by SPINE (7.5KB more) and Titan (just a little more than 2KB). This is mainly due to the need for a more complex management logic in charge of controlling the task-execution engine and to the additional code necessary for coupling the generic C-based architecture to the specific TinyOS components. Nevertheless, the SPINE2 ROM footprint guarantees enough free memory (roughly 7KB) for further framework updates/extensions. Also, if it may be necessary to provide extensions such that the resulting binary code is greater than the available ROM, the modular and reconfigurable architecture of SPINE2 allows for a possible exclusion of some modules at compile-time such as unused sensor drivers or task definitions. On the contrary, on the Z-Stack platform, SPINE2 offers a greater amount of free ROM with respect to SPINE (roughly 10.5KB). Regarding the RAM usage, SPINE2 requires more memory than SPINE on both platforms, but lesser than Titan. It is worth noting that SPINE2 allows developers to allocate at compile-time proper amount of dynamic memory used for instantiating both the task-based application and the buffers necessary for data exchange among tasks. In particular, the amount of RAM shown in Table 3.8 and related to SPINE2 includes a 800-bytes memory space for storing the user-defined applications and a 1600-bytes memory space for dynamic buffer allocation (i.e., in total, more than 2KB of dynamic memory reserved at compile-time).

The energy consumption results have been obtained by running the application benchmark, and adopting a 650mAh Li-ion battery for both sensor platforms. Specifically, the node lifetimes have been experimentally obtained and the related average power consumption has been calculated. SPINE2 shows a little more average power consumption with respect to SPINE due to the more complex runtime engine and to some more information in the SPINE2 messages header which is directly related to the complexity of the distributed task applications.However, these results are acceptable if we consider its more powerful programming abstractions. Notably, SPINE2 shows better results than Titan.

Concerning the bandwidth usage, we have considered not only the application data payload to be transmitted (120 bytes/s, i.e. 20 samples per axis, each of two bytes length) but also both the header of the framework high-level messages and the header of the low-level sensor platform protocol. SPINE2 and SPINE support message fragmentation (see Section 3.2.5 for SPINE2), whereas Titan does not have such a capability. So, we had to redefine the original benchmark application by manually limiting the data buffering to only 10 samples for each data accelerometer axis.

3.3 Using SPINE2

3.3.1 A case study

To demonstrate the effectiveness of SPINE2, in the following section we describe how the SPINE2 task-oriented approach has been adopted for imple-

menting a distributed action recognition system which aims at detecting actions made by individuals, such as walking, kneeling, jumping and so on. Specifically, in Section 3.3.1.1 we first describe the system as reported in [81]. In particular, the authors validate it through Matlab, by delegating the BSN just for data acquisition and transmission to the coordinator, on which all the signal processing tasks are actually performed by means of Matlab. Then, in Section 3.3.1.2 and 3.3.1.3 we provide a real SPINE2 implementation of the same system by moving most of the computation on the sensor nodes, by showing how it is a really straightforward process.

3.3.1.1 The distributed action recognition system

The system, described in [81], makes use of a template matching technique to perform weak classifications from a set of body-worn sensors, whereas an ensemble learning methodology is adopted for achieving the final recognition results.

The functional blocks of the whole system are illustrated in Figure 3.11. The template matching technique uses the Normalized Cross Correlation (NCC) function to compute the similarity of the incoming signals acquired by the sensor nodes (each of which equipped with triaxial accelerometer and biaxial gyroscope) with previously calculated templates of interest (obtained with a supervised learning approach).

The NCC measures the correlation (the range of the results varies from -1 if uncorrelated to +1 if correlated) between a sensor signal time series ($S[x]$) and the template time series ($T[x]$):

$$NCC_{S,T} = \frac{\Delta_S \cdot \Delta_T}{\sigma_S \cdot \sigma_T} \quad , \qquad |NCC_{S,T}| \leq 1 \tag{3.1}$$

where

$$\Delta_S = \sum_{k=1}^{|S|}(S[k] - \bar{S}) \quad , \quad \sigma_S = \sqrt{\sum_{k=1}^{|S|}(S[k] - \bar{S})^2} \tag{3.2}$$

and Δ_T and σ_T are computed similarly to Equation 3.2.

The resulting similarity scores are compared with specific thresholds to weakly classify the incoming signals as true or false, where true indicates that the current body movement generating the signal is classified as a target action (this happens when the signal is similar to the specific template related to the action) while false indicates a non-target action. The previous formulas assume that both the incoming signal and the templates have the same length. If not, a normalization process between S and T is computed as follows:

$$newS(k) = S(\lfloor \frac{|S|}{|T|} \rfloor \times k) \qquad \forall k \in \{1, 2, ..., |T|\} \tag{3.3}$$

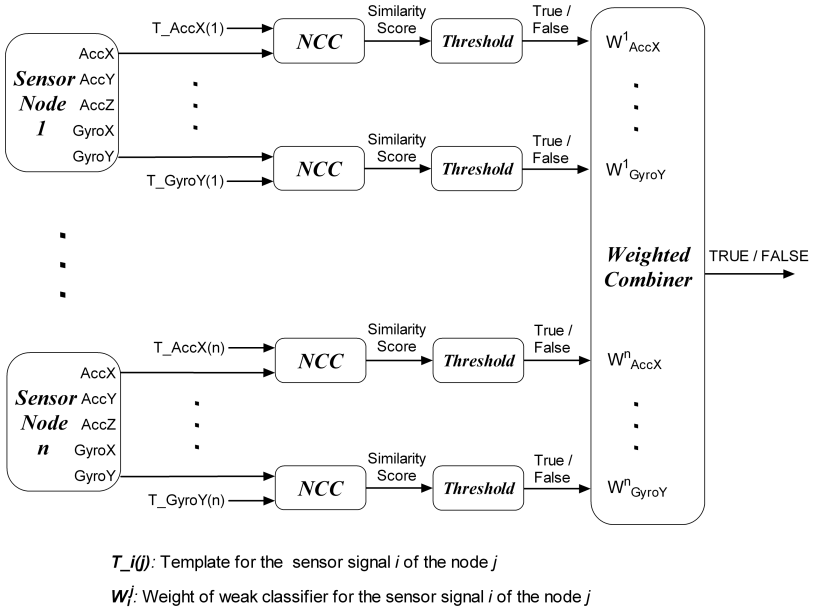


Fig. 3.11. The action recognition diagram blocks.

where T is the longer signal and $newS$ is the transformation of the shorter one.

It is worth noting that, since the approach is based on supervised learning, the templates (a template is related to a specific accelerometer or gyroscope axis of a certain sensor node) are generated during a training phase in which the sensor readings are collected while a person is performing the actions and then manually segmented and labeled. Different template instances related to a same action are extracted so to choose the one that is most similar to the others (comparison are made by calculating the NCC), and thus much more robust to variabilities. Once all the nodes provide the results of these weak classifications, they are aggregated and transmitted to another sensor node (or to the coordinator) which is in charge of weighing and combining them together so to provide a significantly better classification result.

3.3.1.2 SPINE2-based application definition

In the following, the model of the system described in the previous subsection is defined through the SPINE2 modeling language. By focusing on the activities executing on the sensor nodes, it is possible to define the SPINE2 application with the chain of tasks depicted in Figure 3.12. The proper parameter setting for each task, apart from the MergeTask which does not need any specific settings, are also reported.

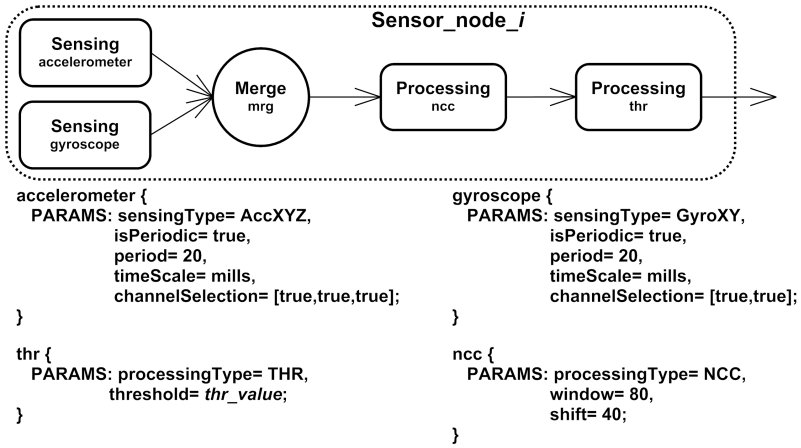


Fig. 3.12. The SPINE2 task chain application instantiated on the sensor nodes.

In particular, the application consists of the following tasks: two SensingTask, for accessing to the on-board accelerometer and gyroscope sensors; a MergeTask which merges the raw sensor streams into a single data stream to be forwarded to the next tasks; two ProcessingTask, for computing the NCC and for comparing results with the threshold values. The sampling time for sensor acquisition is set to 20 ms whereas the ProcessingTask performs a new computation every 40 samples on a data window of 80 samples (i.e. a time window of 1600 ms). These settings have been experimentally determined to be the best trade-off between computational performance and final recognition results related to each of the individual actions taken into consideration.

After defining the per-node application, the implementation of the last stage of the action recognition system, represented by the Weighted Combiner component of Figure 3.11, needs to be performed. One option is to build the Weighted Combiner component into one of the sensor nodes of the network. In this case, further SPINE2 tasks, as shown in Figure 3.13, need to be instantiated on it. In particular, the MergeTask is in charge of collecting all the weak classification results coming from the sensor nodes running the task-based application defined in Figure 3.12, whereas the ProcessingTask is configured for carrying out the final recognition process through a weighted combiner task. Anyway, the node in charge of computing such a weighted combiner may not be necessarily a sensor platform. Thanks to the availability of the BSN coordinator, which is generally a more powerful node, it is possible to perform the last recognition task on it, without any further computational load on the sensor nodes. This has been actually our choice for developing the system, where the Weighted Combiner has been implemented as a Java application running on a PC interfaced to the sensor nodes by means of a

base-station device.

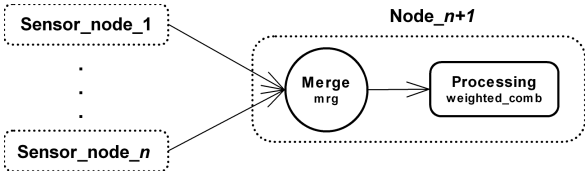


Fig. 3.13. The SPINE2 task chain related to the final recognition phase.

3.3.1.3 From the SPINE2 application model to the real running system

The per-node application shown in Figure 3.12 represents a general application definition implementing a single weak classifier of the action recognition approach depicted in Figure 3.11. Since none of the movements we are interested in needs all the sensor data streams (i.e. data from all sensor channels) for being recognized, a preliminary step is necessary for identifying the actual information aiming at a good final recognition process. Specifically, the needed sensor streams along with their related templates are obtained during such a step and, as described in details in [81], involve the use of four wearable sensor nodes worn on the waist and right calf, thigh, and wrist. Thus, acceleration and angular velocity (three axes of accelerometer and two axes of gyroscope) data are collected from different subjects performing a set of 12 movements (see Table 3.9) several times. Although this raw data comes from real sensors, the steps involved in determining the necessary sensor streams and related templates have been carried out on a PC by means of Matlab. It is worth noting that the actual aim of this preliminary setup phase is to identify the minimum set of weak classifiers that achieve a given recognition sensitivity (or recall) and precision values. As a result, Table 3.9 shows the number of the weak classifiers for recognizing each movement, given three different level of accuracy. To achieve higher accuracy, a larger number of classifiers need to be instantiate over the four sensor nodes. By considering a specific accuracy (i.e. P=90%, R=80%), Table 3.10 shows also the actual sensor streams needed for each movement recognition. The number n in each data stream *Acc*-n* specifies a particular sensor node, i.e. the one on the wrist (1), waist (2), thigh (3), or calf (4). For the sake of brevity, the sensors involved in detecting movements #6, #8, #9, #11, and #12 are not displayed. To be noted that, by excluding these movements, the remaining ones can be easily recognized by means of the analysis of the only accelerometer data.

Table 3.9. Number of weak classifiers needed for detecting movements by considering a precision P and a recall R.

	Movement	N. of classifiers (P=80%, R=70%)	N. of classifiers (P=85%, R=75%)	N. of classifiers (P=90%, R=80%)
#1	Stand to sit	1	1	1
#2	Sit to stand	1	1	1
#3	Sit to lie	1	1	3
#4	Lie to Sit	1	3	3
#5	Bend and Grasp	3	3	5
#6	Kneeling, right leg first	15	15	15
#7	Kneeling, left leg first	3	3	3
#8	Turn clockwise 90 degrees	14	14	17
#9	Turn counterclockwise 90 degrees	5	5	10
#10	Move forward (1 step)	3	3	3
#11	Move backward (1 step)	10	10	10
#12	Jump	14	14	14
	TOT Classifiers	69	71	83

Table 3.10. Number of templates needed for movements detection (with precision=90% and recall=80%).

	Movement	Sensor stream needed	N. weak classifier
#1	Stand to sit	AccZ-3	1
#2	Sit to stand	AccZ-3	1
#3	Sit to lie	AccZ-1, AccY-2, AccZ-4	3
#4	Lie to Sit	AccY-2, AccZ-3, AccY-4	3
#5	Bend and Grasp	AccX-1, AccY-1, AccY-2 AccZ-2, AccZ-3	5
#6	Kneeling, right leg first	-	15
#7	Kneeling, left leg first	AccX-1, AccZ-1, AccY-4	3
#8	Turn clockwise 90 degrees	-	17
#9	Turn counterclockwise 90 deg.	-	10
#10	Move forward (1 step)	AccZ-1, AccZ-2, AccY-3	3
#11	Move backward (1 step)	-	10
#12	Jump	-	14

3.3.1.4 Performance analysis

In this subsection, some performance tests are shown to understand if the computing and memory capabilities of the sensor nodes can guarantee an effective and efficient real-time data processing. In particular, the memory usage analysis of the application and the execution time of the ProcessingTask related to the NCC function (the one having the highest computational cost) have been carried out on both TelosB motes running TinyOS and the custom platform based on Z-Stack.

3.3.1.4.1 Memory usage analysis

As previously described, in the template matching method, each signal acquired by the on-board sensors has to be compared with a set of pre-calculated

templates, each of them related to a specific movement. Such templates are required to be stored on the sensor nodes and specifically in the RAM as the slow access time of the flash memory may represent a bottleneck during the real-time processing. Thus, the amount of free available memory for storing templates is a key limiting factor determining the maximum number of movements the system is able to recognize at the same time.

The RAM memory usage of the SPINE2 application on the TinyOS and Z-Stack sensor platforms (i.e. the implementation of the weak classifier) is, respectively, 4280 KB and 6289 KB, whereas the free available one is, respectively, 5960 KB and 1903 KB. In particular, the RAM usage includes the platform operating system (i.e. TinyOS or Z-Stack), the SPINE2 middleware and the task instances of Figure 3.12. Moreover, the differences with respect to the Table 3.6 are because of a different amount of RAM reserved at compile-time as dynamic memory space. Moreover, the smaller amount of free memory in the custom sensor node is due to the lesser total RAM available (see Table 3.4) and to the more memory required by Z-Stack compared to TinyOS.

On the basis of the free memory, in Table 3.11, the maximum number of templates that can be stored on the sensor nodes are shown. A sampling time $ST = 20ms$ for data acquisition has been considered whereas a fixed length for all the pre-calculated templates is assumed. This is reasonable because the template matching approach includes the normalization process (see Equation 3.3) in case of different length between incoming signal and template.

Table 3.11. Maximum number of templates (at a sampling time of 20ms) and NCC ProcessingTask average execution time.

	Template length [samples]	Template duration [ms]	Max number of templates	NCC average exec. time [ms]
TelosB	40	800	74	5.32
	60	1200	49	7.00
	80	1600	37	8.57
	100	2000	29	9.38
Z-Stack-based node	40	800	23	2.96
	60	1200	15	3.75
	80	1600	11	4.50
	100	2000	9	4.91

For our tests, we obtained templates of 1.6s length (i.e. 80 samples) because of two main reasons. First, most of the movements have an average duration between 1s and 1.5s. Thus, we set the NCC to compute the correlation result on a window size of 80 samples (see Figure 3.12). Secondly, this is the best trade-off between storage and processing capabilities. In fact, if we used smaller templates describing all the movements, each nodes would have performed a normalization process for each weak classifier before computing the NCC, with a significant increase in the computation load.

3.3.1.4.2 Processing execution

The analysis of the completion time for the ProcessingTask performing the NCC is shown in the following. This is an important performance index for understanding the feasibility of the system to obtain an efficient real-time movement recognition. In Table 3.11 the NCC-task execution time is shown by varying the length of the signals in terms of samples (or window size W).

For evaluating the actual performance of the system in managing the recognition of all the movements defined in Table 3.9, we have to consider that the computation of the m weak classifiers on each node has to be performed on every new S data samples (S is the shift parameter of the ProcessingTask usually set to $W/2$, see Figure 3.12). The adopted sampling time is $ST = 20ms$, so the total amount of time taken to execute m classifiers should be less than the time for acquiring S new samples:

$$m \cdot T_{NCC}(W) \leq ST \cdot S \tag{3.4}$$

where $T_{NCC}(W)$ denotes the execution time of Table 3.11.

By considering the node of the BSN in charge of managing the greatest number of weak classifiers ($m = 24$), Table 3.12 reports the total amount of execution time by varying the W parameter. On the basis of the provided results, it is rather clear that the ProcessingTask implementing the NCC performs very well on both sensor platforms.

Table 3.12. Total execution time for computing m=24 weak classifiers [ms].

W and S=W/2 [samples]	TelosB $m \cdot T_{NCC}(W)$	Z-Stack Custom-platform $m \cdot T_{NCC}(W)$	$ST \cdot S$
40 - 20	127.68	71.04	400
50 - 25	147.84	79.68	500
60 - 30	168.00	90.00	600
70 - 35	184.80	96.96	700
80 - 40	205.68	108.00	800
100 - 50	225.12	117.84	1000

3.3.2 SPINE2 for implementing "Virtual Sensor"

Signal processing for WSNs usually comprises of multiple levels of data abstraction, from raw sensor data to data calculated from processing steps such as feature extraction and classification.

In this section, a multi-layer task model based on the concept of Virtual Sensors to improve architecture modularity and design reusability. Virtual Sensors are abstractions of components of WSN systems that include sensor sampling and processing tasks and provide data upon external requests. The Virtual Sensor model implementation relies on SPINE2. The proposed model

is applied in the context of gait analysis through wearable sensors. A gait analysis system is developed according to a SPINE2-based Virtual Sensor architecture and experimentally evaluated.

3.3.2.1 The WSN-oriented Virtual Sensor Architecture

Physical sensors map an observed physical quantity, such as temperature, acceleration, or sound, onto a data value and produce an output. The output is generated when inputs change, as the result of an event, or in response to a (timed) request. Physical sensors are transducers converting values from one form to another using physical processes. Signal processing algorithms convert values using digital processes. This observed similarity is the motivation behind the virtual sensor abstraction. Every processing task can be represented as a virtual sensor. Therefore, if we consider a complete WSN system, we can model its data processing part as a multi-level hierarchy of virtual sensors as shown in Figure 3.14. Moreover, virtual sensors may be implemented directly in a programming language, or as networks of already existing virtual sensors.

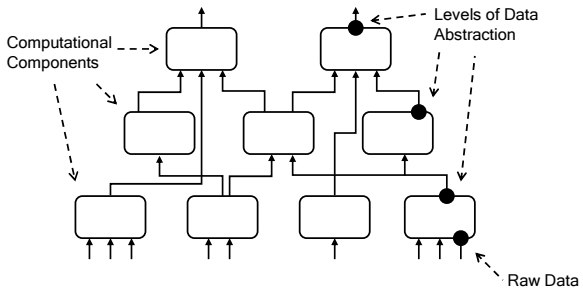


Fig. 3.14. Multi-layer Signal Processing

Figure 3.15 shows the defined WSN-oriented virtual sensor system architecture. A user requests certain outputs given specified inputs. This request is handled by the Virtual Sensor Manager, which configures a set of virtual sensors to handle the computational task. Virtual sensors use the Buffer Manager to setup communication through the use of efficient buffers. Once configured, the system is activated, and virtual sensors cooperate to produce the final outputs.

3.3.2.1.1 Virtual Sensor Definition

Software frameworks are usually introduced to provide programmers with abstractions to isolate them from low-level implementation details. Virtual sensors provide a new level of abstraction at the software level by allowing signal processing tasks to be defined and composed easily. Furthermore, VS

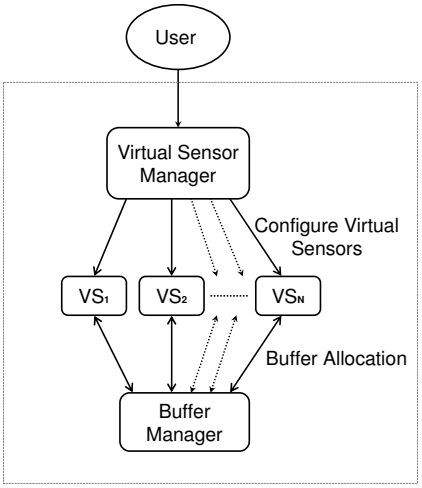


Fig. 3.15. BVS Architecture.

abstractions allow signal processing tasks to be modified or changed at design or runtime without affecting the rest of the system. In Figure 3.14 every component represents a processing task applied to a stream of data originated from physical sensors and can be modeled as a virtual sensor. The output of each virtual sensor is defined by a set of inputs and its configuration. More formally, a virtual sensor i, denoted as VS_i, is defined as:

$$VS_i = \{I_i, O_i, C_i\} \tag{3.5}$$

where I_i denotes the set of inputs, O_i denotes the set of outputs, and C_i denotes the configuration of VS_i. The configuration of each virtual sensor defines the type of its inputs and outputs, the particular implementation used for a given computational task, and a set of parameters required for a particular implementation. In particular, C_i is defined as:

$$C_i = \{\mathbf{t}_{in}, \mathbf{t}_{out}, d, p\} \tag{3.6}$$

where $\mathbf{t}_{in}$ is a vector that describes the types of inputs I_i, $\mathbf{t}_{out}$ is defined similarly for the outputs O_i, d represents the specific VS implementation, and p denotes the VS configuration parameters. In particular, if the user does not specify d, the Virtual Sensor Manager (described below) will select the implementation.

This definition provides high modularity for application design. In fact, different configurations of the same virtual sensor can be easily substituted without requiring changes in the rest of the design. This property therefore enables a component-based approach for application development in which an application is assembled out of well defined components appositely interconnected. Moreover, it can be used when environmental changes require a

new implementation of a particular signal processing component for a given application. Alternative implementations do not need to be loaded into main memory at all times. They can be stored in flash memory, or transferred over the air upon request.

VS can be further composed to create higher-level VS. This allows to define multiple abstraction levels that capture the successive processing and interpretation of sensor data and system components that perform data fusion. High-level VS identify abstractions that are useful to support code modularity and reusability. In fact, if an implementation of a VS is replaced with another one, where one or more VS components are changed but the interface is the same, there is no need to change the rest of the system.

More formally, the composition of n Virtual Sensors to form an higher-level VS can be defined as follows:

$$VS^* =< VS_1, VS_2, ...VS_n >= \{I^*, O^*, C^*, L\} \tag{3.7}$$

where $I^* \subseteq I_1 \cup I_2 ... \cup I_n$, $O^* \subseteq O_1 \cup O_2 ... \cup O_n$, $C^* = \{C_1, C_2, ...C_n\}$, and L is the set of links connecting outputs and inputs of $\{VS_1,...,VS_n\}$

3.3.2.1.2 Virtual Sensor Manager

Once all virtual sensors are configured, no additional control is required during execution. However, configuration requires significant support from the Virtual Sensor Manager (VSM). The VSM is responsible for creating and configuring virtual sensors and connections among virtual sensors. The following subsections will describe the main functionalities of the VSM (virtual sensor configuration and overall system configuration).

Virtual Sensor Configuration: the current configuration of a virtual sensor may be invalidated by changes in its inputs or connections with other virtual sensors, therefore reinitialization could happen at any time. For example, Figure 3.16(a) describes a system that takes a temperature reading in Fahrenheit, and a heart rate in beats per minute. In Figure 3.16(b) a new thermometer, that produces output in Celsius, is introduced. VS_1 has to be reconfigured to handle such change. To be able to configure/reconfigure the system at run time, the VSM manages a table that maps each available combination of possible inputs and outputs to the appropriate virtual sensor implementation. This can be represented by the set A. Each entry $a \in A$ is defined as:

$$a = \{\mathbf{t}_{in}, \mathbf{t}_{out}, \psi\} \tag{3.8}$$

where ψ is a particular virtual sensor implementation. If the modification is not drastic enough to require changing the virtual sensor implementation, reconfiguration can alter parameters of a given implementation. During the configuration of a virtual sensor, VSM includes the address of the selected virtual sensor implementation and the required configuration parameters.

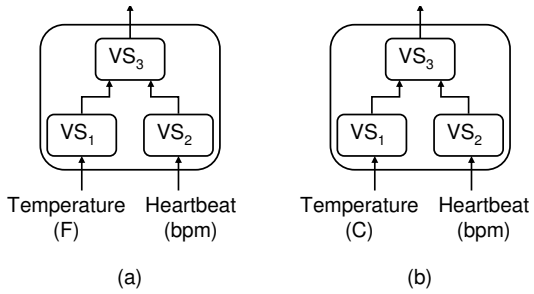


Fig. 3.16. Example of Input Modification in Virtual Sensors.

Overall System Configuration: while individual virtual sensors do not hold any information about other virtual sensors, the overall system relies on their cooperation. At the beginning of the system execution, the VSM receives the VS topology configuration graph. Based on the requirements of the topology configuration, the VSM initializes the appropriate VSs and connects them as required. Input and output types are a property of each virtual sensor. An output of one of the virtual sensors can also be an input of another virtual sensor. For example, in Figure 3.17, configuration of VS_3 and VS_4 depends on the input they receive from VS_1. To simplify the configuration and reconfiguration process, the VSM initializes VSs in a specific order, to meet the requirement that each virtual sensor cannot be created until all inputs are configured. This ordering can be determined with a topographical sort of the topology configuration graph.

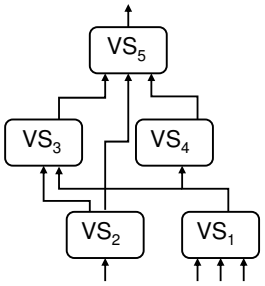


Fig. 3.17. Example of Input/Output Dependency in Virtual Sensors.

Buffer Manager: signal processing for WSNs often relies on combining data from multiple sources and locations. As a result, virtual sensors can have multiple inputs from different sensor nodes. To avoid synchronization issues, virtual sensors implicitly use buffers for communication. The Buffer Manager (BM) controls dynamic buffer allocation and manages data flow in the system.

When a virtual sensor is created and configured, it initiates a data buffer for its output. The virtual sensor contacts the BM and requests the creation of a buffer sufficient to hold its output. The BM allocates a circular buffer of the required size and returns the `bufferID`. This `bufferID` is propagated by the VSM to other virtual sensors that are interested in data of this particular buffer. To read from a buffer, a virtual sensor must register with the buffer as a reader, specifying the number of samples it can consume at a time. Every time the producer writes to the buffer, the BM checks if the buffer has enough information for any of the readers, and signals them when they can access the data. Figure 3.18 shows an overview of the BM operation. In particular, it shows that BM keeps track of buffers by ID, tracking the point where the producer (e.g. W) is writing to, and where each individual reader (e.g. R_1, R_2) is reading from. If the producer VS is reconfigured, and its output is changed, the BM removes the buffer that is associated with the previous output and initiates a new buffer, based on the new configuration information.

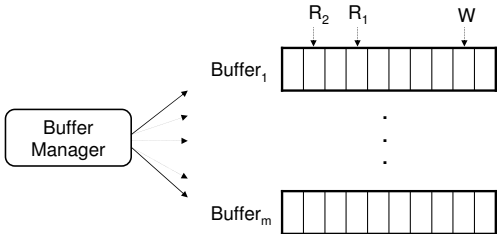


Fig. 3.18. Buffer Manager Overview.

3.3.2.2 Implementing Virtual Sensors in SPINE2

The Virtual Sensor architecture described in Section 3.3.2.1 is straightforwardly implemented through the SPINE2 framework. In Figure 3.19 basic conversion schemas for the translation of Virtual Sensors into SPINE2 task-oriented applications are shown. In particular, only simple (flat) virtual sensors have been taken into consideration as it is quite intuitive to translate a virtual sensor defined as composition of flat virtual sensors.

In the most simple case, a virtual sensor defined as a basic functional block incorporating some kind of operation on its single input can be translated into a SPINE2 *data-processing task* (see Figure 3.19(a)). In fact, a generic SPINE2 data-processing task (such as *dpTask*) is defined as a functional component having a single input and a single output, differently from the data-routing task. Obviously, the operations that have to be performed by the task (specified by its configuration) depend on the actual functionalities of the virtual sensor. If the virtual sensor does not have a generic input but raw data (such

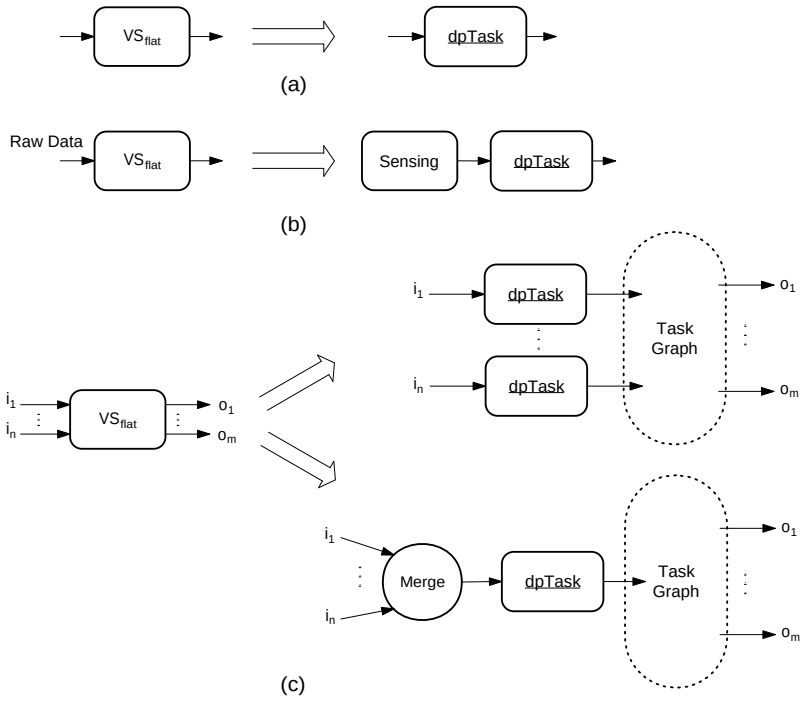


Fig. 3.19. Translation of Virtual Sensors into SPINE2 task-oriented models.

as data coming form an hardware sensor on a wireless node), the corresponding translation includes the introduction of a SensingTask, specifically configured for representing the digital data source (see Figure 3.19(b)). Finally, Figure 3.19(c) shows the translation of a simple virtual sensor having multiple inputs and outputs. In this case, the corresponding SPINE2 tasks can be configured in several ways on the basis of the actual definition of the virtual sensor and of the type description of its inputs/outputs. In particular, two different translations are shown. In the first one, we have a single data-processing task for each input. These tasks, along with the not-specified Task Graph, carry out the overall computational operation performed by the virtual sensor. Conversely, in the other translation, inputs are merged by a single data-routing task (namely, the MergeTask) and provided to a generic task graph. In either case, the more complex the function defined for the virtual sensor gets, the more complex the set of actual interconnected tasks would be. Of course, there could exist a more generic SPINE2 translation in which some of the inputs merge on an MergeTask, the other ones become inputs of data-processing tasks.

It is worth noting that the two application modeling abstractions, virtual sensors and tasks, have strong similarities. In fact, both of them enables creation of applications in a modular and easily reconfigurable way by using

elementary functional blocks (virtual sensors or tasks) which do not have any functional couplings with each others. This is due to the fact that they have no knowledge of the provenance of their inputs nor the destination of their outputs.

3.3.2.2.1 A Gait Virtual Sensor

As a test application for virtual sensors in SPINE2, we implemented a virtual sensor-based system for automatic event annotation based on a left-right Hidden Markov Model (HMM) [82]. The HMM associates each data sample with a state. States typically consist of multiple data samples and the transition from one state to another starts with an event which consists of a single sample. By training the model, we can identify these key events within a movement. Walking being a cyclical activity, events and states repeat over a period of time.

The HMM annotation system consists of four parts:

1. Sampling of acceleration data from physical sensors. We used LIS3LV02DQ MEMS based accelerometer from ST Microelectronics for the data collection. The data is collected at a sampling frequency of 20 Hz which is sufficient for slow movements like walking. The data is copied into the buffer pool allocated by the Buffer Manager as it gets collected.
2. Pre-filtering of sampled data. The main purpose of pre-filtering is to remove any noise from the original data. The first stage of pre-filtering involves calculating the 5-point moving average for removing high frequency noise. This data is then normalized by subtracting 100-point mean and dividing by 100-point standard deviation to get the final pre-filtered data.
3. Extraction of features from filtered data. The pre-filtered data stream is processed by the feature extractor and the first derivative, second derivative and the existence of any peaks is calculated [82]. These three features, along with the prefiltered data sample is passed on to the next virtual sensor.
4. HMM-based annotation of events from the extracted features. The annotation is done based on the probability of occurrence of a state. This requires training of the model to create the tables for mapping the features to the probability of observing a state. In addition, we also consider the probability of a state transition. These two tables are generated based on training which was done in MATLAB.

The system uses a single sensor node, and only one implementation was created for each virtual sensor.

The Gait Virtual Sensor is defined as a linear interconnection of four virtual sensors as described in Figure 3.20. Arrows connecting virtual sensors denote the data flow from the producer to the consumer. The initial request is generated by the SPINE coordinator running on a PC. This request initiates the configuration of each of the four virtual sensors. Once all the virtual

sensors have been initialized, the HMM virtual sensor starts producing output every sample. The sampling interval is defined for the Accelerometer VS, which acts as the data source for the whole system.

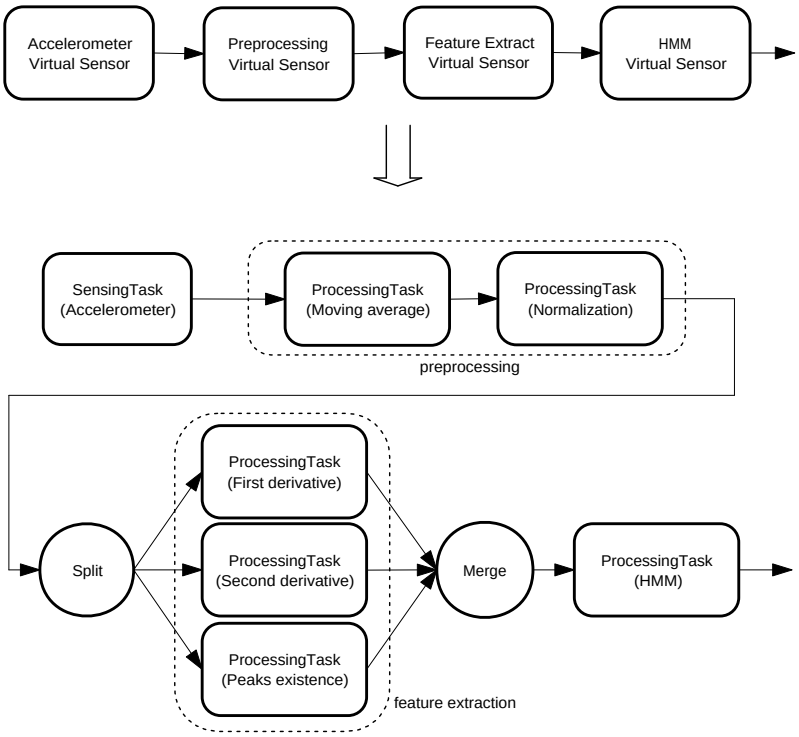


Fig. 3.20. GAIT analysis Application.

As described in Section 3.3.2.2, each Virtual Sensor can be implemented through one or more SPINE2 tasks, on the basis of its actual complexity. First of all, the presence of a Data Raw source, namely the accelerometer, requires the use of a SensingTask configured to acquire data from a real onboard sensor with proper settings (like the sampling rate). The Preprocessing VS, which performs preliminary computational functions can be translated into two ProcessingTasks sequentially interconnected with each of them in charge of carrying out a specific elementary operation. In a similar way, the Feature Extract VS can be decomposed into three further ProcessingTasks, individually devoted to compute a specific feature extraction function on the data coming from the previous preprocessing step. Since all these tasks require the same data, a Split data-routing task has to be used. As soon as all the features are computed, they have to be made available to the HMM ProcessingTask. This task, representing the actual implementation of the HMM VS,

receives its input data from the Merge data-routing task, which is in charge of collecting the feature extraction results.

In the following subsections, we first describe some implementation details about GVS and other software modules implementing the same gait analysis functionality but developed in C and Matlab; we then analyze the obtained results and compare them with the results obtained with the C and Matab software modules.

3.3.2.2.2 Implementation

Figure 3.20 shows the schemas of the GVS according to the higher level model and its translation based on SPINE2. In particular, each Virtual Sensor can be implemented using one or more SPINE2 tasks, on the basis of its actual complexity.

We started with a MATLAB model described in [82]. The code was simplified to match sensor node capabilities, e.g. using fixed-point math with limited precision. Based on this code, a version of the code was developed in C, and the outputs were compared. The model was retrained based on the updated code. After making changes to assure matching output between the C and MATLAB code, the C code was adapted to SPINE2 and ported to nesC for an ad-hoc implementation on the sensor nodes. This allowed us to compare the effectiveness of using SPINE2 in the development of the virtual sensor with respect to the use of a lower level programming language like nesC.

The main challenge of adapting MATLAB models to SPINE and C implementation was the lack of hardware floating point multipliers in the microcontrollers of the wearable units, particularly MSP430 in our platform. Floating point operations are handled in software and are computationally expensive. Hence, we chose to approximate floating point operations in the original model with integer operations for our implementation. This technique appeared to be effective while from time to time generated inconsistencies due to the conversion error. Furthermore, when the range of floating point numbers were large, this technique would be less effective as it has to represent numbers with a large number of integer bits. To reduce the occurrence of these cases, we had to perform further optimization on the signal processing algorithm to reduce the ranges or reduce the effect of error due to conversion on the outcome of the algorithm.

Although the layered model of the defined virtual sensors allowed us to develop and verify one component at a time, one of the major limitation that we faced during the development was debugging. The development platforms available for TinyOS offer limited support for debugging. Due to this reason, the functionality testing and debugging was done using the C version. However, problems that we faced after porting the code on to the mote were much more serious and hard to debug. Problems like stack overflow due to excessive memory usage were extremely difficult to detect. We also faced issues due to high packet loss hindering the basestation-mote communication. Another major problem was the freezing of system due to higher processing overhead of

one component over the others. This also caused interruptions in the communication. Another problem was the occurrence of "impossible" state transitions in the HMM. The root cause of the problem was determined as overflows. The HMM involved a large number of summations which periodically overflowed, leading to unpredictable results. Figuring out these problems required in-depth analysis of the code and careful optimizations and corrections on appropriate components. Reducing the memory footprint of the application was one of the major challenges that we faced during the whole development.

3.3.2.2.3 Analysis of Results

For our test case, we used this system to extract heel-down and heel-lift events from a walking subject. The sensor node contained a single tri-axial accelerometer sampling at 20 Hz (the highest sampling rate achievable while performing the processing steps). The event annotation was initially quite sensitive to sensor node misplacement, so we trained it with data from one subject with ten different trials. Each trial contained approximately 50 steps and had a slightly different sensor placement. This significantly increased the accuracy. The sensor node performed annotation and broadcast the raw samples so results between different implementations could be compared.

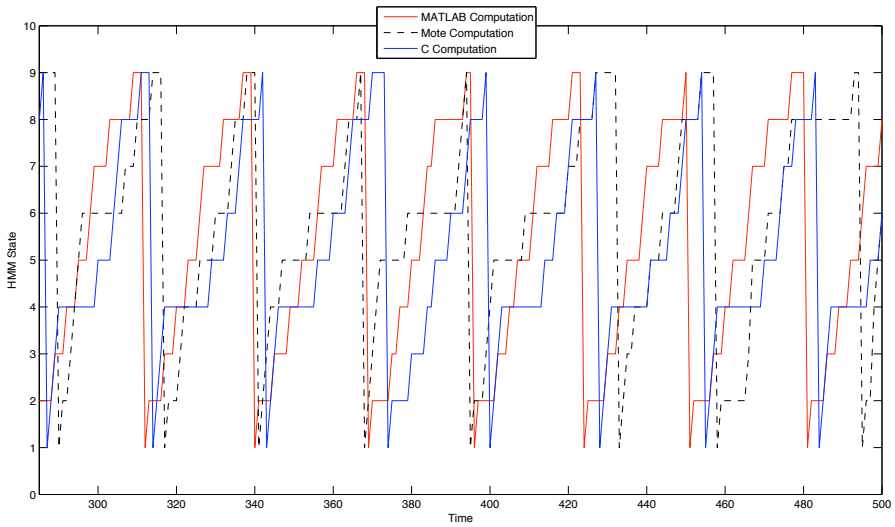


Fig. 3.21. Comparison of Matlab, C, and SPINE2 implementations of the Gait Virtual Sensor.

As can be seen in Figure 3.21, the versions produce similar results. Most of the events were accurately predicted by our implementation barring the slight offsets occurring at times. There is a drastic difference observed between the

sensor node output and the others on the far right side of Figure 3.21. Also, the state transitions were not completely synchronized. For example, in the actual MATLAB implementation, state marked 6 is always momentary where as this does not happen in the other implementations. This is due to the fact that the model tries to catch up after a wrong state calculation. Consequently, the total time taken for a complete sequence of transitions remained almost the same for the whole experiment (on average, the difference is less than 10 ms). This time, which can also be defined as the time taken to reach the same state in the next cycle is termed as the stride time. Table 3.14 shows the average stride time measured for the three implementations.

Table 3.13. Event Annotation Results for Different Implementations.

Implementation	Stride Time (Smpl)	Stride Time (s)
MATLAB 28.21		1.41
C (PC) 28.04		1.40
SPINE2 28.04		1.40

3.4 SPINE-*

Introducing autonomic properties into a sensor network software can be made in many different ways and with different levels of details. Nevertheless, we believe that the best practice is to clearly separate the business logic of the application from the specific autonomic management operations. Such a separation of concerns can lead to a significant benefit: application programmers can concentrate their efforts on defining the purposes and the characteristics of an application without taking care of anything else, whereas the autonomic behavior can be easily added later without affecting the previously defined application logic.

The benefit from using the SPINE2 architecture is that its task-oriented abstraction perfectly matches with the need for a mechanism assuring the isolation and composition properties. Moreover, the addition of the autonomic features into SPINE2 did not involve any deep changes in the node-side part of the SPINE2 architecture depicted in Figure 3.4, since it was sufficient to define a set of new tasks specifically conceived for providing the necessary autonomic functionalities to the task-based applications.

In this section, the autonomic architecture of the SPINE-* framework is described.

3.4.1 Architecture

The basic abstraction of the proposed autonomic architecture is depicted in Figure 3.22. Specifically, it is composed of two distinctive planes, one for the

user application and one for providing the autonomic operations. Each plane is constituted by a set of SPINE2 tasks, each of which providing a specific activity and defined independently from all other tasks. This means that a task is only aware of its input data but it does not care about which particular task has provided such a data. This implies that a same type of task may be used both in the application plane and in the autonomic plane (i.e. there exist two different task instances of a same task type), on the basis of specific needs.

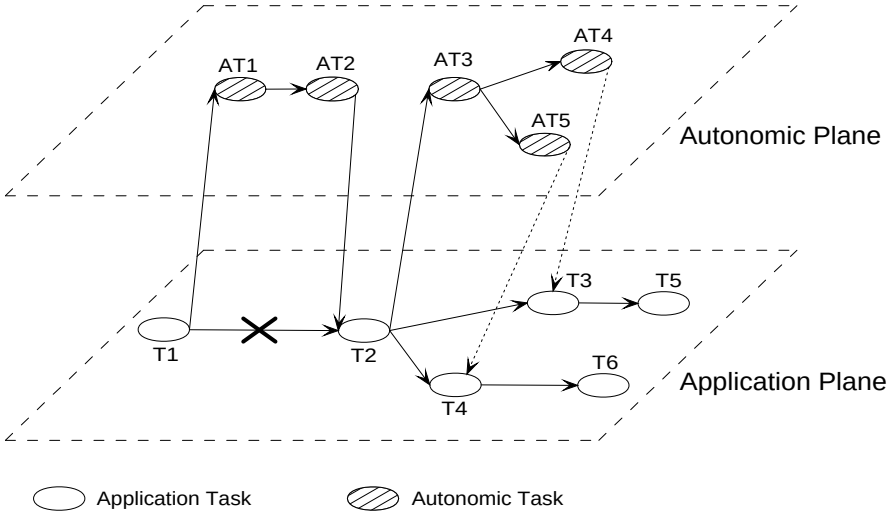


Fig. 3.22. The autonomic architecture with the two interacting planes.

When the autonomic plane needs to execute on the basis of the data flowing inside the user application or to perform a reconfiguration on the application tasks, a kind of interaction between the two planes is established. Despite this interaction, the separation of concerns property still holds. In fact, the application continues to have the same behavior with no awareness of the presence of the autonomic plane, whose post-addition allows to introduce self-* behavior.

For example, supposing that the autonomic tasks AT1 and AT2 are conceived to perform some kind of operation on the output data of T1 prior to redirecting them to task T2, the direct connection T1-T2 has been substituted by the interposed tasks AT1 and AT2. In this case, T2 continues its execution independently from the actual input data provenience. When the interaction consists of some kind of reconfiguration action, the application structure remains substantially the same. For example, depending on data coming from T2, the remaining part of the autonomic plane may trigger some operations aimed at changing the execution parameters of tasks T3 and T4 (the dot-

ted arrows are for differentiating the configuration connection from the data connection).

The proposed autonomic architecture is able to address the common situations for satisfying the main self-* properties, as discussed in the following.

Self-configuration. A useful property for a WSN application is to reconfigure task parameters at runtime on the basis of some dynamic system changes. Two examples are depicted in Figure 3.23. In Figure 3.23(a), depending on the output data of the Processing task (which performs some kind of processing on the raw sensed data), the AutoSensReconfig task, belonging to the autonomic plane, may change the application behavior by acting on the Sensing task parameters, i.e. by changing its sampling rate or even by disabling/enabling its execution. Similarly, as shown in Figure 3.23(b), the ReconfigTask task may perform some changes in either the sensing or the processing task parameters (or both together) as a consequence of an external request. In particular, the component triggering the reconfiguration is not the task-based application itself but the basestation-side application which, as part of the autonomic WSN, may require a different sensor data processing on the basis of dynamic changing conditions.

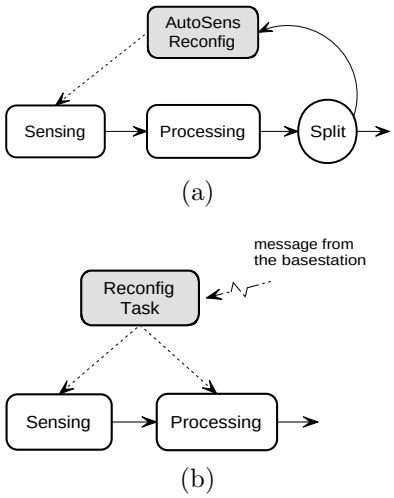


Fig. 3.23. Self-configuration examples.

Self-healing. If we consider a sensor network applied to the human body for health-care applications the requirement of recovering from faults and errors become an important and actually critical issue. Then, self-healing may be considered an essential characteristic a sensor network should incorporate for assuring its reliability and correctness. In Figure 3.24 an autonomic plane is interposed between the Sensing and the Processing so to guarantee the

quality of raw data and then to avoid that a data corruption could affect the entire application correctness. In particular, the DataFaultsDetection task is in charge of detecting possible sensor reading faults and deciding if a data filtering operation (performed by the Filtering task) is required before processing them. Solutions for correctly recognizing some common specific data faults can be found in [47].

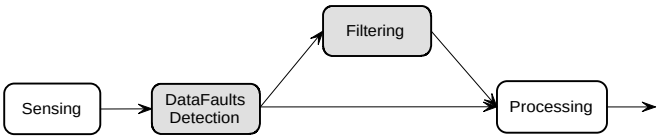


Fig. 3.24. Self-healing examples.

Self-optimization. In order to extend the operating lifetime of a WSN application, the energy consumption has to be specifically taken into consideration. Data transmission (the most energy consuming operation) should be therefore avoided when it is not strictly necessary for the purposes of the application or when a processing activity has encountered an error so that its results are incorrect and have to be discarded. As shown in Figure 3.25, a possible solution consists in defining a specific ResultsVariability autonomic task that is capable of recognizing the variability of processing results over the time. Only in case such a variability exceed a certain threshold, results have to be sent to destination, otherwise they are discarded.

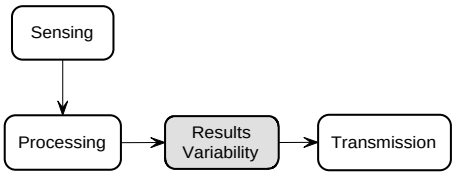


Fig. 3.25. Self-optimization example.

Self-protection. The security problem in the context of WSN is becoming an increasingly important issue as the spread of real and information-critical solutions are becoming relevant. In particular, the privacy of information transmitted in a sensor network is one of the most priority goal. In Figure 3.26, the Encrypting task is conceived to encrypt data coming from the Processing task when outdoor activities are recognized. This allows to adapt the application on the basis of its execution context and at the same time avoiding that the intensive-computing encrypting task is executed when not necessary, i.e. when the WSN system is running at home.

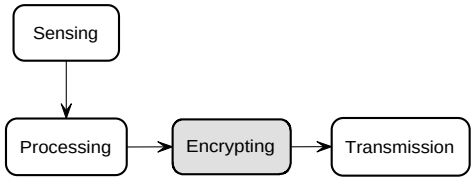


Fig. 3.26. Self-protection example.

3.4.2 A use case: autonomic activity recognition

The proposed SPINE-* approach has been put at test by turning an existing SPINE2 activity recognition application (as described in [83]) into an autonomic one.

The system consists of a desktop application, implemented in Java on the coordinator, and two node-side applications designed by following the programming abstraction approaches provided by SPINE2. The desktop application is responsible for gathering pre-elaborated data taken from the accelerometer sensors of two nodes, placed on the waist and on a leg of a person, and relies on a K-Nearest Neighbor classifier for recognizing postures (i.e. lying, sitting or standing still) and a movement (walking) defined in a training phase. The computation on each node-side application consists in processing specific features (mean, max and min functions) on the acquired accelerometer raw data, and merging and transmitting results to the coordinator.

The implementations developed by using SPINE2 is effective for the final results of the system but, unfortunately, it does not provide some important key features required by a WSN application. For instance, such solution is not, in fact, aware of the quality of data coming from sensor readings. What is worse, it is not possible to modify the application behavior on the basis of changes in the environmental context. Thus, no adaptation or correctness checking operations can be assured during node-side application execution.

The addition of an autonomic plane can overcome all these limitations by allowing to transparently define the necessary operations for effectively supporting self-* application management. In particular, the resulting autonomic task-graph with the two different planes is shown in Figure 3.27.

The DataFaultsDetection autonomic task is able to recognize possible corrupted data. If no corruption is detected, data are directly forwarded to the Split task. If data corruption is detected and no recovering operation can be performed, data are discarded because they are not useful for the application purposes and further computation is avoided. Finally, if the detected data faults can be solved, data are forwarded to the Filtering task for recovering data before further computation. The ReconfigTask logic has been conceived for listening possible request from the desktop application requiring a different parameters setting for the Sensing task aiming to disable the sampling operations for a certain amount of time or to decrease its sampling rate. This

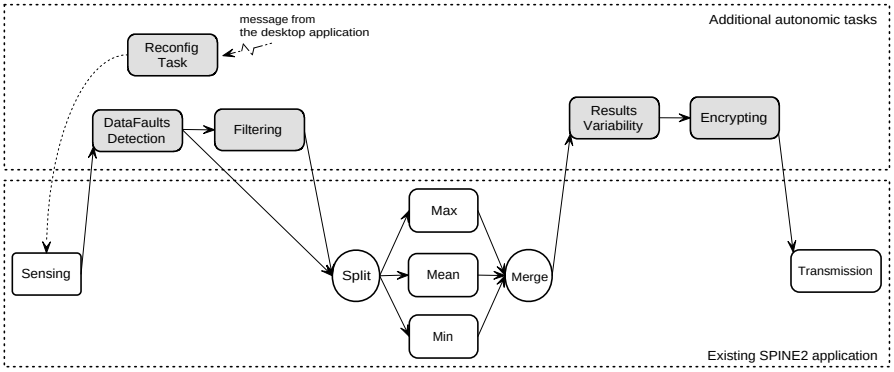


Fig. 3.27. Autonomic Activity Recognition.

is useful when the desktop application recognizes that the monitored person is not moving or is slowly moving, and a very frequently data processing and transmission is not necessary. The ResultsVariability task can analyze features results and recognize if they exhibit a certain level of invariability, or smooth variability. In this case the task discards them because it is useless to transmit unchangeable processing results to the basestation. Finally, the Encrypting task encrypts the computed data before their transmission to the coordinator.

In Table 3.14 the application accuracy related to four activities is shown. As it can be noted, the results are quite good demonstrating that the application is not affected by the computational overhead introduced by the autonomic plane.

Table 3.14. Activity recognition classification accuracy.

	Walking	Sitting	Standing	Lying
SPINE-∗	98.5%	97.3%	97.8%	98.3%

3.5 Embedded self-healing layer for detecting and recovering sensor fault

Avoiding erroneous behavior of WSN-based systems is an issue of fundamental importance, especially for critical health-care applications. In this regard, proper self-healing techniques should be able to fulfill requirements such as fault tolerance and reliability by detecting, and possibly recovering, faults and errors at runtime.

If we consider a sensor network applied to the human body for health-care applications the requirement of detecting and possibly recovering from faults and errors become an important and actually critical issue. Thus, the global quality of an application not only depends on how well it has been designed and implemented to accomplish its specific purposes but also on how well it can deal with problems at runtime. Since it is not reasonable for a WSN to be manually and constantly maintained and supervised after deployment, the system itself needs to incorporate specific self-managing capabilities. For such reasons, autonomic characteristics may be incorporated into a WSN [4] and, in particular, self-healing techniques may be considered as an essential characteristic a sensor network should incorporate for assuring its reliability and correctness.

In particular, we first analyze how faults in the sensing readings can affect the quality of a WSN application (i.e. a human activity recognition one), and then we show how a self-healing layer, capable of detecting and possibly recovering such faults, can improve the recognition accuracy. Specifically, the self-healing layer is part of SPINE-*, which aims at easily and explicitly defining the autonomic behavior by reusing the same high-level task-oriented paradigm adopted for the application logic definition. We demonstrate by experimental results that SPINE-* can greatly improve the effectiveness of an application, and we find that the defined autonomic elements provided can mitigate faults affecting the sensor data so empowering system behavior correctness. We conclude that the use of self-healing capabilities in WBSNs is a critical requirements for improving the reliability of applications, especially the ones more sensitive to the quality of data.

3.5.1 The testbed: activity recognition application

Since in this section we concentrate on faults that can potentially affect the raw data acquired by sensors, we need a proper reference application on which performing some data quality tests. The BSN application taken into consideration for our analysis is a human body activity recognition system able to recognize some postures (e.g. lying, sitting or standing still) and a movement (e.g. walking) of a person [64, 84].

The system consists of a Java-based application running on the coordinator-side (i.e. a PC to which a sensor node is connected by acting as a base-station bridge), and two wearable nodes (one placed on the waist and the other on the thigh of the right leg) based on the Tmote Sky platform[8] equipped with a custom sensor board having a 3-axis accelerometer. The activity recognition system relies on a K-Nearest Neighbor (KNN)-based classifier, executing on the coordinator, that takes the most significant features performed on the accelerometer data and recognizes the movements defined in a training phase. In particular, the nodes run two different applications, both designed through

[8] http://www.snm.ethz.ch/Projects/TmoteSky

the task-based programming abstraction approach provided by SPINE2, each of which performing specific features extraction on different accelerometer channels, as illustrated in Figure 3.28.

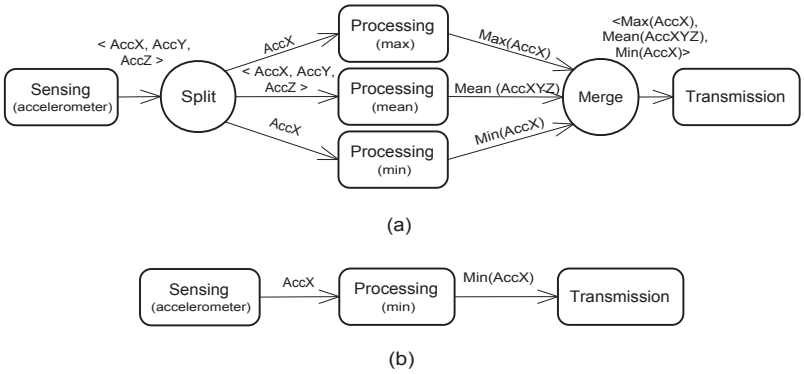


Fig. 3.28. The SPINE2-based node-side application on the waist node (a) and on the thigh one (b).

For evaluating the impact of data faults on the previously described application, we have carried out a testbed by considering a pre-defined sequence of postures/movements represented by the state machine reported in Figure 3.29. In particular, each activity state lasts roughly 30 seconds, with a sensor sampling time of 25ms, whereas the features (Min, Max and Mean) are computed on a windows of 40 sampled data every new 20 samples (shift) acquired by the sensors. The training phase uses a KNN-based classifier parameterized with K=1 and the Manhattan distance which performs quite well as classes (lying down, sitting, standing still and walking) are rather separate. Accordingly to such settings and considering the correctness of the sensor raw data, the obtained classification accuracy over the whole activities transition pattern is 99.75%.

In the following subsections, we analyze the variation in the classification accuracy when the original accelerometer data streams (whose traces are depicted in Figure 3.30) are affected by specific faults. It is worth noting that from now on we will consider only the accelerometer streams actually involved in the computation, i.e. the X, Y, Z channels of the waist node and the X channel of the thigh node (see Figure 3.28).

Thus, for evaluating the effects of faults with respect to the original experiment we have carried out several tests after having altered the original data traces by artificially injecting faults corresponding to each of the models identified in [47]:

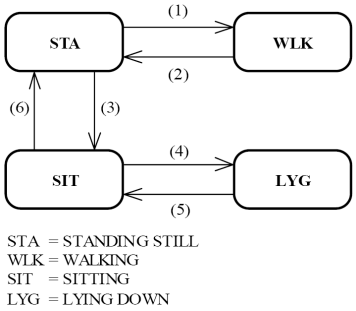


Fig. 3.29. The state transitions for the adopted testbed.

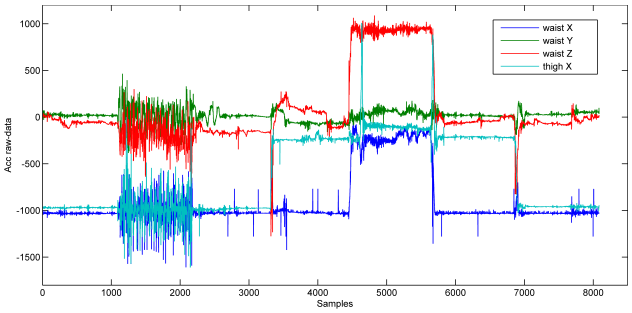


Fig. 3.30. Original accelerometer raw-data streams.

- *Short* faults are random spikes in the data trace, representing random irregularities in the sensor readings.
- *Noise* is an increased variance in a continuous region of a data streams.
- *Constant* faults can be represented either by a arbitrary and fixed value not correlated to the real phenomenon under observation or by a region in the data trace whose values are biased by a certain value.
- *Accumulative* faults, i.e. drift behavior in the sensor readings, which is commonly represented by a monotonic distancing from the actual values.

3.5.1.1 Short faults

The short faults are modeled as spikes disseminated on the P percentage of the data trace and whose single value is determined by the original sensor reading multiplied by a factor C, representing the fault intensity parameter.

An example of data traces affected by short faults, with parameters P=5% and C=3, is depicted in Figure 3.31. As it can be noted, faults are uniformly and randomly injected over all the trace of each accelerometer channel, and this represents the worst case with respect to the same previously defined parameters.

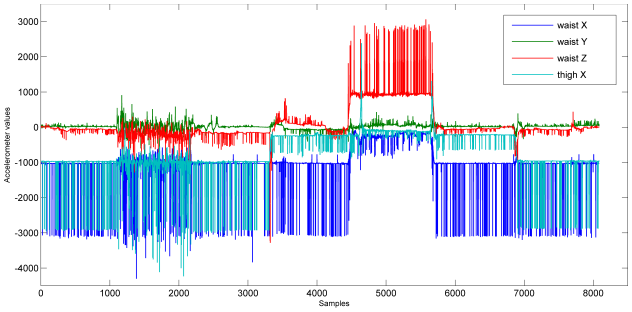


Fig. 3.31. Raw-data streams affected by short faults (*P=5%, C=3*).

In Table 3.15 the results of the classification accuracy are reported by considering short faults affecting all channels and by varying the percentage of the faults. As shown, the presence of just 1% of samples affected by faults get the accuracy to greatly decrease at less than the 80%. Much worse accuracy results are obtained if we consider more frequent faults over the traces.

Table 3.15. Accuracy results with short faults over all channels and with *C=3*.

Channel	P	Accuracy
All	1%	79.90%
All	5%	55.09%
All	10%	51.86%
All	25%	48.14%
All	50%	46.65%

In Table 3.16, accuracy results are displayed by taking into consideration only one channel affected by faults. It is rather clear that the accuracy is much more affected by the presence of short faults on the X channel of the waist node and especially the X channel of the thigh node.

Table 3.16. Accuracy results with short faults on a single channel and with *C=3*.

Channel	P	Accuracy
Waist X	1%	98.25%
Waist Y	1%	99.75%
Waist Z	1%	99.75%
Thigh X	1%	81.63%
Waist X	5%	96.26%
Waist Y	5%	99.75%
Waist Z	5%	99.75%
Thigh X	5%	44.91%

3.5.1.2 Noise faults

Since noise faults are represented by prolonged increasing variance in the readings of a sensor, a Gaussian distribution can be adopted for deliberately injecting such kind of faults. In particular, we consider a normal random generator having as mean the original specific samples value to be replaced, and a parameter σ as its standard deviation. Moreover, with the parameter N we identify the number of random regions to affect on a single data trace, whereas K represents the length of such regions, measured in samples number.

An example of all channels affected by noise faults, with parameters σ=300, *N=1* and *K=1000*, is depicted in Figure 3.32.

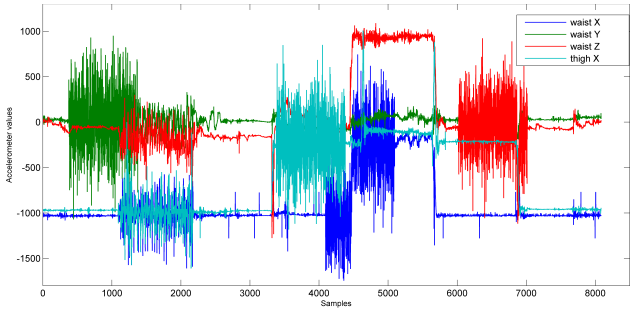


Fig. 3.32. Raw-data streams affected by noise faults (*σ=300, N=1, K=1000*).

In Table 3.17 the results of the classification accuracy are reported by considering noise faults affecting all channels and by varying the σ parameters. As it can be noted, although the considered noisy region is rather wide (1000 samples, i.e. 25 seconds) in all channels, the activity recognition system continues to yield a good classification accuracy, even when the standard deviation has very high values.

Table 3.17. Accuracy results with noise faults over all channels by varying the σ parameter (*N=1, K=1000*).

Channel	σ	Accuracy
All	100	98.64%
All	300	91.77%
All	500	89.54%
All	1000	88.29%
All	1500	84.85%

All the previous considerations are confirmed by the results shown in Table 3.18, where at a fixed standard deviation, the accuracy results are above the 90% unless the region length is greater than 1000 samples.

Table 3.18. Accuracy results with noise faults over all channels by varying the noise region length (*σ=500, N=1*).

Channel	K	Accuracy
All	100	98.54%
All	200	96.88%
All	500	94.14%
All	1000	91.17%
All	2000	81.00%

3.5.1.3 Constant faults

As already described, constant faults can be represented either by an invariant repetition of an arbitrary value that is uncorrelated to the observed phenomenon or by a region in the trace biased by a certain constant value. In the following we will consider only the second kind of constant faults, since it is more general. In particular, our model includes three parameters: C, representing the bias constant affecting the sensor readings, N, representing the number of random regions to affect on a single data trace, and K, that is the length of such regions, measured in samples number.

An example of data streams affected by constant faults, with parameters *C=500*, *N=3* and *K=600*, is depicted in Figure 3.33.

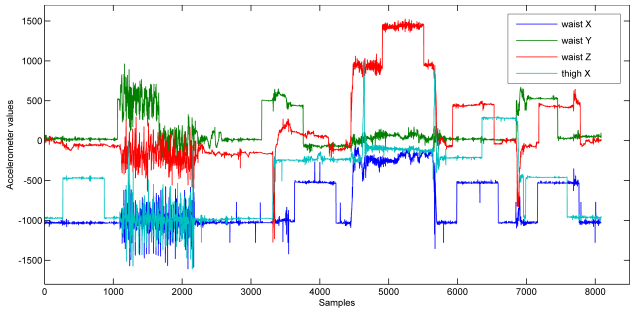


Fig. 3.33. Raw-data streams affected by constant faults (*C=500, N=3, K=600*).

In Table 3.19 and Table 3.20 the results of the classification accuracy under constant faults, by varying the regions length and the bias value, are reported.

In the first case, the accuracy remains above 90% with faults affecting regions having no more than 500 samples (i.e. 12.5 seconds), whereas in the second table the accuracy remains above 80% despite the bias value can be more than the maximum variation (roughly 1000) of the original data traces.

Table 3.19. Accuracy results with constant faults over all channels by varying the region length (*C=500, N=3*).

Channel	K	Accuracy
All	100	98.51%
All	200	95.24%
All	500	91.06%
All	1000	82.08%
All	2000	56.65%

Table 3.20. Accuracy results with constant faults over all channels by varying the bias value (*N=3, K=500*).

Channel	C	Accuracy
All	100	99.75%
All	300	95.87%
All	600	84.41%
All	1000	81.97%
All	1500	80.49%

3.5.1.4 Accumulative faults

As anticipated earlier, the accumulative fault manifest itself as a drift behavior in the sensor readings, and can be modeled as a monotonic distancing from the actual data values. This is the worst case of data faults, since, in real situations, it is mostly due as a consequence of a permanent fault in the sensor hardware.

An example of a single data stream (Y channel of the waist node) affected by accumulative faults is depicted in Figure 3.34.

The results of the classification accuracy under accumulative faults affecting different accelerometer channel are shown in Table 3.21. Similarly to the results obtained with short faults (see Table 3.16), the accuracy is much more affected by the presence of accumulative faults on the X channel of both nodes.

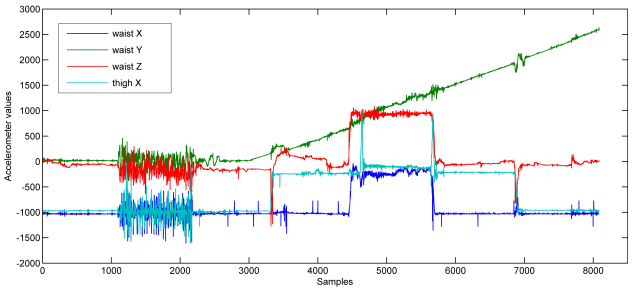


Fig. 3.34. Accumulative faults on the Y data stream of the waist node.

Table 3.21. Accuracy results with accumulative faults affecting different accelerometer channels.

Channel	Accuracy
Waist X	38.46%
Waist Y	99.00%
Waist Z	99.25%
Thigh X	55.83%
All	20.34%

3.5.2 The self-healing plane: empowering fault tolerance of activity recognition through SPINE-*

Although SPINE-* is able to address the common situations for satisfying the main self-* properties (i.e. self-configuration, self-healing, self-optimization, and self-protection), we are now interested in its self-healing capabilities. In particular, for enhancing our activity recognition application, the autonomic tasks layer depicted in Figure 3.35 have been developed and tested. Such an autonomic plane is interposed between the Sensing and the Processing tasks so to improve the quality of sensor data and then to mitigate data corruption that could affect the application correctness. The DataFaultsDetection task is in charge of detecting possible sensor reading faults and deciding if a data filtering operation (performed by the Filtering task) is required (if possible) before processing them.

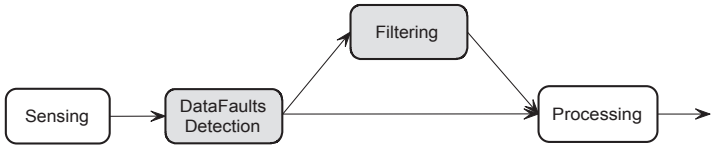


Fig. 3.35. Self-healing tasks in SPINE-*.

For the sake of space, in the following we will only describe the approach for detecting and recovering the short faults by showing the obtained results.

3.5.2.1 Short faults detection and recovery results

Since short-faults are represented by irregular spikes over part or whole of the data traces, the idea under the detection and recovery procedures is to analyze the variability of the sensor readings along each of the accelerometer data streams before passing them to the processing tasks in charge of performing the necessary features extraction.

By considering a consecutive region of samples, the mean and the standard deviation are computed over such a data window. Afterward, for each single sensor reading inside this region, its value is compared to the previously computed standard deviation. If it is much greater, then it is likely to be a fault. Once a data fault is detected, the recovery approach adopted in our experiments consists in replacing its original value with the one of the previous sampled data.

Thus, the main parameters of this detection approach are the data window length W, on which mean and standard deviation (sd) are computed, and the threshold T, multiple of the standard deviation, to which the sample values are compared.

The cleaned accelerometer data traces as a result of executing the previously described approach on the corrupted data streams of Figure 3.31 are shown in Figure 3.36. The obtained result is very similar to the original data streams (see Figure 3.30). Specifically, a window $W=40$ and a threshold $T=3*sd$ have been adopted.

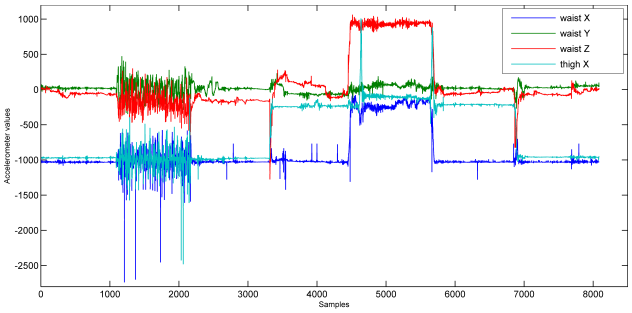


Fig. 3.36. Raw-data streams recovered from short-faults generated with $P=5\%$, $C=3$.

A comparison of the classification accuracies before and after faults detection/recovery are reported in Table 3.22. The adopted approach for recovering the accelerometer data streams is really effective in case the frequency of short

faults is less then 25% for each channels. Much lower accuracy improvements are obtained in the other cases. This is due to the fact that if faults are very frequent, they cannot be well distinguished from the correct data trend, i.e. it is impossible to determine whether a specific value corresponds to a behavior derived by correct sensing operations of the sensor or it should be identified as a fault.

Table 3.22. Accuracy improvements with short-faults over all channels and with *C=3*.

P	Accuracy (affected data)	Accuracy (recovered data)
1%	79.90%	99.75%
5%	55.09%	99.75%
10%	51.86%	98.51%
25%	48.14%	59.55%
50%	46.65%	47.64%

3.6 MAPS: an agent-based programming framework for WSNs

In this section, an innovative programming framework, MAPS, for developing agent-based applications on WSNs is presented. MAPS (Mobile Agent Platform for Sun SPOT) has been specifically designed for WSNs based on the Java-based sensor platform Sun SPOT[9] and allows to model mobile agents by means of a multi-plane state machines driven by ECA (Event-Condition-Action) rules. Its architecture is based on component interacting through events and supporting agents with a minimal set of services including message transmission, agent creation, agent cloning, agent migration, timer handling and easy access to the sensor node resources (sensors, actuators, input switches, flash memory and battery).

A modified lightweight version of MAPS, called TinyMAPS, is also briefly described. While MAPS is specifically conceived for higher processing-capable sensor devices, Sun SPOTs, TinyMAPS is its adaptation tailored for more constrained platforms such as Sentilla JCreate[10].

In the following, we first discuss on the use of agents in WSNs and the requirements for developing efficient Multi-Agent Systems (MASs) on resource-constrained platforms. Afterward, the design, implementation and experimentation of MAPS and TinyMAPS are described. Finally, a comparison with

[9] http://www.sunspotworld.com
[10] http://sentilla.tomcoh.com/developers/perkpage

other agent-based programming frameworks and a case study for realtime human monitoring are also illustrated.

3.6.1 Requirements for MAS development on WSNs

Since the mobile agent-based paradigm has fully demonstrated its effectiveness in conventional distributed systems as well as in highly dynamic distributed environments, there are good motivation to prove that they can also effectively deal with the programming and management issues that WSNs have posed. As a confirmation, in the last years agent technology has been successfully used in WSNs at different levels (application, middleware, network) [85]. In particular, the main agent-oriented research efforts have been to date devoted to the following WSN research themes: network routing, data dissemination and fusion, in-network coordination, programming frameworks, high-level system architectures and applications.

Although several research efforts have demonstrated that mobile agents are a suitable computing paradigm for supporting the development of distributed applications, services, and protocols on WSNs, the development of flexible and efficient MASs remains a challenging and very complex task due to the currently available resource-constrained sensor nodes. In the following, we first discuss the use of agents in the context of WSN on the basis of the Lange and Oshima research work [86], which delineates seven good reasons for using agents in traditional networks. We then provide an outline on the requirements for MAS development over WSNs.

3.6.1.1 On the use of mobile agents for WSN applications

In their seminal paper [86], Lange and Oshima advertised at least seven good reasons for using mobile agents in generic distributed systems. In the following we describe them with reference to the WSN context.

1. *Network load reduction.* Mobile agents are able to access remote resources, as well as communicate with any remote entity, by directly moving to their physical locations and interacting with them locally to save bandwidth resources. A mobile agent incorporating data processing capabilities can migrate to a sensor node, perform the needed operations on the sensed data and transmit the results to a sink node. This is more desirable, rather than a periodic transmission of raw sensed data from the sensor node to the sink node and the computation of data processing on the latter.
2. *Network latency overcoming.* An agent provided with proper control logic may move to a sensor/actuator node to locally perform the required control tasks. This overcomes the network latency that will not affect the real-time control operations also in case of lack of network connectivity with the base station.

3. *Protocol encapsulation.* If a specific routing protocol supporting multi-hop paths should be deployed in a given zone of a WSN, a set of cooperating mobile agents encapsulating this protocol can be dynamically created and distributed into the proper sensor nodes without any regard for standardization matter. Also in case of protocol upgrading, a new set of mobile agents can easily replace the old one at run-time.
4. *Asynchronous and autonomous execution.* These distinctive properties of mobile agents are very important in dynamic environments like WSNs where connections may not be stable and network topology may change rapidly. A mobile agent, upon a request, can autonomously travel across the network to gather needed information "node by node" or to carry out the programmed tasks and, finally, can asynchronously report the results to the requester.
5. *Dynamic adaptation.* Mobile agents can perceive their execution environment and react autonomously to changes. This behavioral dynamic adaptation is well suited for operating on long-running systems like WSNs where environment conditions are very likely to change over time.
6. *Orientation to heterogeneity.* Mobile agents can act as wrappers among systems based on different hardware and software. This ability can well fit the need for integrating heterogeneous WSNs supporting different sensor platforms or connecting WSNs to other networks (like IP-based networks). An agent may be able to translate requests coming from a system into specific suitable requests to submit to another different system.
7. *Robustness and fault-tolerance.* The ability of mobile agents to dynamically react to unfavorable situations and events (e.g. low battery level) can lead to a better robust and fault tolerant distributed systems; e.g. the reaction to the low battery level event can trigger a migration of all executing agents to an equivalent sensor node to continue their activity.

An interesting taxonomy about WSNs and their relationships with multi-agent systems can be found in [87]. In particular, the major motivation of using agents over such networks is that many WSNs properties are shared with and can be actually supported by agents and multi-agent systems: physical distribution, resource boundedness, information uncertainty, large scale, decentralized control and adaptiveness. Moreover, as sensors in a WSN must typically coordinate their actions to achieve system-wide goals, coordination among dynamic entities (or agents) is one of the main features of multi-agent systems. In the following, the aforementioned common properties are discussed:

- Physical distribution implies that sensors are situated in an environment from which they can receive stimuli and act accordingly, also through control actions aiming at changing their environment. Situatedness is a main property of an agent and several well-known agent architectures were defined to support such important property.

- Boundedness of resources (computing power, communication and energy) is a typical property both of sensor nodes as single units and of the WSN as a whole. Agents and related infrastructures can support such limitation through intelligent resource-aware, single and cooperative behaviors.
- Information uncertainty is typical in large-scale WSNs in which both the status of the network and the data gathered to observe the monitored/controlled phenomena could be incomplete. In this case, intelligent (mobile) agents could recover inconsistent states and data through cooperation and mobility.
- Large scale is a property of WSNs either sparsely deployed on a wide area or densely deployed on a restricted area. Agents in multi-agent systems usually cooperate in a decentralized way through highly scalable interaction protocols and/or time- and space-decoupled coordination infrastructures.
- In large-scale WSNs, centralized control is not feasible as nodes can have intermittent connections and also can suddenly disappear due to energy lack. Thus, decentralized control should be exploited. The multi-agent approach is usually based on control decentralization transferred either to multiple agents dynamically elected among the available set of agents or to the whole ensemble of agents coordinating as peers.
- Adaptiveness is the main shared property between sensors and agents. An agent is by definition adaptive in the environment in which is situated. Thus, modeling the sensor activity as an agent or a multi-agent system and, consequently, the whole WSN as a multi-agent system, could facilitate the implementation of the adaptiveness properties.

3.6.1.2 Requirements and issues

Although the agent paradigm has great potential to help the development of WSN applications, as demonstrated by all the previously discussed motivations, it is quite clear that the development of MASs for WSNs requires not only the same efforts required by conventional distributed systems but also the fulfillment of additional requirements specific to WSNs [88].

A direct exploitation of generic software agents into sensors is not so trivial, since research in traditional multiagent systems domain does not take into consideration the presence of severe resource constraints that typically arise on sensor nodes. In fact, the management of poor computational and energy resources, leading to many technical limits in designing a practical WSN mobile-agent middleware system, represent the most critical challenge in such a network. Moreover, research often does not opportunely consider that communication might be slow and intermittent and that nodes might be unreliable and failure prone.

Since software agents generally exhibit intelligent behaviors for autonomously coordinate their actions to achieve specific system-wide goals, the complexity of a middleware infrastructure for managing and supporting such agents'

properties must be carefully considered with respect to the available resources. This is an extremely important issue because a fundamental requirement is to achieve a good execution performance on each single node for guaranteeing global efficiency and scalability.

Based on our experiences and on the results in literature, we truly believe that for facing with the resource-constrained problem, an agent-based system has to be defined following some design requirements:

- The MAS architecture server must be as lightweight as possible, which implies the avoidance of heavy concurrency models and, therefore, the exploitation of cooperative concurrency control mechanism to run agents.
- A plug-in-like components organization is recommended, in order to dynamically and selectively activate services that are needed while deactivating the useless ones for improving the overall system performance.
- The agent structure must be also lightweight so that agents can be efficiently executed and migrated. This not necessarily implies that agents cannot show complex and intelligent behaviors, but simply that the mechanisms for defining and encoding their behavioral models have to be simple so that the architecture devoted to agent control and execution is not so resource-hungry.
- Mobile agents must be natively characterized on the basis of the functional layer to which they belong: application, middleware and network layer. They must be also able to locally interact to enable cross-layering.

Despite the actual effectiveness of the aforementioned guidelines, the efforts required for developing efficient MASs may fairly vary on the basis of the features that each single sensor platform provides to the developers.

When possible, the limited resources problem can be overcome by executing heavy software agents, encapsulating computational-intensive functions, into external devices, e.g. components with higher processing capabilities residing outside the WSN. This makes it necessary to properly design and implement MASs by providing the necessary capabilities for allowing a closer interaction between WSNs and traditional networks and distributed systems. If it is not possible to rely on components with much more computational resources, MAS developers have to consider that the execution of advanced agent-based applications is limited by the use of low-overhead techniques and algorithms which necessary have to sacrifice optimality and accuracy, so that agents have to behave as best as possible given the available node resources.

Platform features heterogeneity also brings to an incomparable computing capabilities leading to difficulties in designing a common MAS architecture that could be suitable and efficient for execution on different sensor types. In particular, the need to address such a challenge will occur very soon since WSNs are expected to be deployed into a growing ubiquitous environment, so it is not unlikely to suppose that interaction among different typologies of networks will be a common situation. Although MAS research for WSN has

traditionally considered homogeneous sensor node architectures, this assumption is too restrictive if we think about next generation WSN applications.

Another important issue in developing MAS concerns their architecture design approach and related agent definition primitives, so to opportunely satisfy WSN application developers requirements. Most of the research efforts conducted so far are based on a bottom-up approach. On the basis of the sensor nodes hardware, research focuses on how to provide proper software abstractions to assist application agent designers in defining common or advanced tasks, without requiring to deal with low-level details for hardware and networking management. Making the control part of the agents more expressive is the way for achieving a simplification of the agent design while keeping a rapid WSN application re-programming. This approach can also guarantee a reduction of the agent code size (and consequently a general better migration performance), because most of the macro-functionalities are already implemented into the MAS middleware and are directly accessible to agents on each WSN node running the middleware. A problem of such a solution is that often the available high-level abstractions may not be suitable for many applications that necessitate of a more fine-grained control of the node resources, which is generally needed for defining much more efficient tasks. However, although a fine-grained task control is ideal for reaching a more program execution flexibility, it can lead to a potentially bigger, and consequently error-prone, agent code. The alternative, i.e. the top-down approach, is basically based on a first deep understanding of what the primary application requirements are, which become the main driver for the design of the agent-based middleware. In this case, the provided agent programming constructs may not be so straightforward to use, and even may be very application-specific oriented so that a more general use is not possible.

3.6.2 State-of-the-art and Related Work

Generally speaking, MASs support mobile agents by basically providing an agent server, an API for mobile agent programming and, sometimes, by supporting programming and administration tools. In particular, the agent server is able to execute agents by providing them with basic services such as migration, communication and resource access. In the last decade, a significant number of MASs for IP-based distributed computing systems have been developed. The majority of them are Java based (e.g. Aglets, Voyager, Ajanta, JADE etc.) and few others rely on other languages (DAgents, ARA etc.). In the context of WSNs it is challenging to develop MASs for supporting mobile agent-based programming [89], due to the currently available resource-constrained sensor nodes, and very few real systems have been to date proposed and concretely implemented. In the following, we first describe the most significant available research prototypes based on TinyOS operating system, and then, we introduce the Java-based agent programming frameworks.

Agilla [90] is an agent-based middleware developed on TinyOS and supporting multiple agents on each node. Agilla provides two fundamental resources on each node: a tuplespace and a neighbor list. The tuplespace represents a shared memory space where structured data (tuples) can be stored and retrieved, allowing agents to exchange information through spatial and temporal decoupling. A tuplespace can be also accessed remotely. The neighbor list contains the address of all one-hop nodes needed when an agent has to migrate. Agents can migrate carrying their code and state, but they cannot carry their tuples locally stored on a tuplespace. Packets used for node communication (e.g. for agent migration/cloning, remote tuple accessing) are very small to minimize messages losses, whereas retransmission techniques are also adopted.

In [91] the authors propose an extension of Agilla to support direct communication based on messages. In particular, to establish direct communications, agents are mediated by a middle component (named landmark) that interacts with agents through zone-based registration and discovery protocols.

ActorNet [63] is an agent-based platform based on the Actor model. To overcome the difficulties in allowing code migration and interoperability due to the strict coupling between applications and sensor node architectures, actorNet exposes services like virtual memory, context switching, and multitasking. Thanks to these features, it effectively supports agent programming by providing a uniform computing environment for all agents, regardless of hardware or operating system differences. The actorNet language used for high-level agent programming has syntax and semantics similar to those of Scheme with proper instruction extension.

In [92] another mobile agent framework is proposed. The framework is implemented on Crossbow MICA2DOT motes. In particular, it provides agent migration and agent interaction based both on local shared memory and network messages.

The above described MASs for WSNs [90, 91, 63, 92] are all implemented for TinyOS-based sensor platforms and use *ad-hoc* languages for agent programming (e.g. Agilla uses a micro-programming language, whereas actorNet employs a functional-oriented language). Although some supported operations (e.g. migration) are very efficient, programming complex tasks is not so straightforward and, moreover, developers need to learn another very specific language. The Java language, through which Sun SPOT and Sentilla JCreate sensors can be programmed, due to its object-oriented features, could provide more flexibility and extendibility for an effective implementation of agent-based platforms. Currently, the only available Java-based mobile agent platforms for WSNs (apart from MAPS and TinyMAPS presented in this thesis) are AFME [60] and MASPOT [93].

The AFME framework, a lightweight version of the agent factory framework purposely designed for wireless pervasive systems and implemented in J2ME, has been recently ported onto Sun SPOT and used for exemplifying agent communication and migration in WSNs. AFME is strongly based on

the Belief-Desire-Intention (BDI) paradigm, in which agents follow a sense-deliberate-act cycle. In AFME, agents are defined through a mixed declarative/imperative programming model. The declarative Agent Factory Agent Programming Language (AFAPL), based on a logical formalism of beliefs and commitments, is used to encode an agent's behavior by specifying rules that define the conditions under which commitments are adopted. The imperative Java code is instead used to encode perceptors and actuators. However, AFME was not specifically designed for WSNs and, particularly, for Java Sun SPOT.

MASPOT is a mobile agent system natively designed for Sun SPOTs that, differently from the other Java-based MAS, is able to provide agent's code migration, since it does not rely on the Isolate-based mechanism. In particular, both weak and strong migration are supported. The type of migration is defined for each agent at creation time and cannot change during the agent life cycle. The MASPOT inter-agent communication is based on the tuple spaces model, similar to the one adopted by Agilla. Communication between the base station and the mobile agents requires support for agent mobility. Such a communication is basically established by means of a chain of references from the base station to the node where an agent currently is. When an agent moves to a new node, it leaves behind a marker indicating the next node to which it has migrated. Furthermore, a specific procedure exists to eliminate circular chains of references that will no longer be used.

3.6.3 MAPS Architecture and Programming Model

MAPS is a novel Java-based framework for wireless sensor networks based on Sun SPOT technology which enables agent-oriented programming of WSN applications. The Sun SPOT sensor platform (currently supported by Oracle) is based on the IEEE 802.15.4 communication standard and on the Squawk VM which is fully Java compliant and CLDC1.1-compatible. MAPS has been appositely defined for resource-constrained sensor nodes according to the following requirements:

- *Component-based lightweight agent server architecture* to avoid heavy concurrency models and, therefore, exploit cooperative concurrency to execute agents.
- *Lightweight agent architecture* to efficiently execute and migrate agents.
- *Minimal and pluggable core services* involving agent migration, agent naming, agent communication, activity timing, sensor resource capability access (actuators, input signalers, flash memory, and battery).
- *Plug-in-based architecture extensions* through which any other service should be defined in terms of one or more dynamically installable components implemented as single or cooperating (mobile) agent/s.
- *Java as programming language* for the agent system and (mobile) agents.

In the following subsections we focus on the system architecture and the agent programming model.

3.6.3.1 System architecture

The MAPS architecture (see Figure 3.37) is based on components that interact through events and offer a set of services to mobile agents including message transmission, agent creation, agent cloning, agent migration, timer handling, and easy access to the sensor node resources.

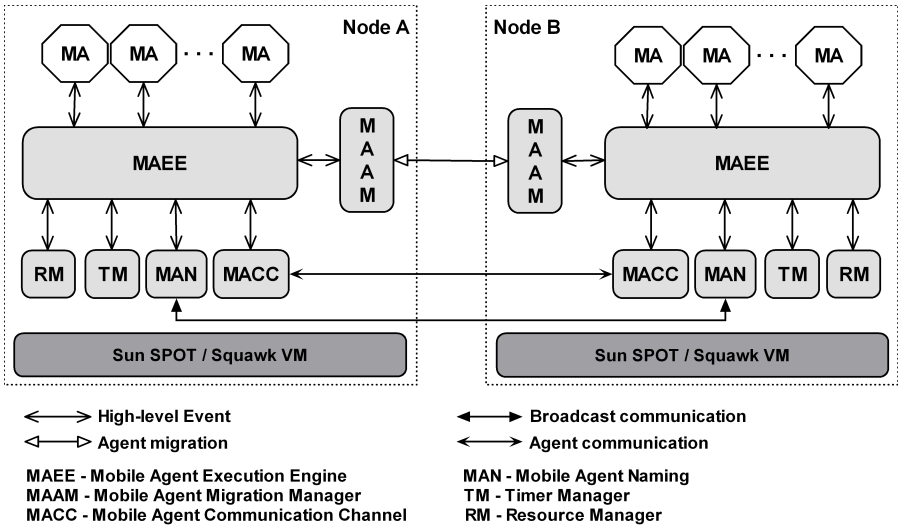


Fig. 3.37. MAPS architecture.

In particular, the main components are:

- *Mobile Agent (MA)*, which is the basic high-level component defined by user for constituting agent-based applications.
- *Mobile Agent Execution Engine (MAEE)*, which manages the execution of MAs by means of an event-based scheduler enabling lightweight concurrency. MAEE also interacts with the other service-provider components to fulfill service requests (message transmission, sensor reading, timer setting, etc) issued by MAs.
- *Mobile Agent Migration Manager (MAMM)*, which supports agents migration through the Isolate hibernation/dehibernation feature provided by the Sun SPOT environment. The MAs hibernation and serialization involve data and execution state whereas the code should already reside at the destination node (this is a current limitation of the Sun SPOTs which do not support dynamic class loading and code migration).

- *Mobile Agent Communication Channel (MACC)*, which enables inter-agent communications based on asynchronous messages (unicast or broadcast) supported by the radiogram protocol.
- *Mobile Agent Naming (MAN)*, which provides agent naming based on proxies for supporting MAMM and MACC in their operations. MAN also manages the (dynamic) list of the neighbor sensor nodes that is updated through a beaconing mechanism based on broadcast messages.
- *Timer Manager (TM)*, which manages the timer service for timing MA operations.
- *Resource Manager (RM)*, which enables access to the resources of the Sun SPOT node: sensors (3-axial accelerometer, temperature, light), switches, leds, battery, and flash memory.

3.6.3.2 Agent programming model

In Figure 3.38 the Mobile Agent model is depicted. In particular, the dynamic behavior of MA is modeled as a multi-plane state machine (MPSM). The GV component represents the global variables, namely, the data inside an MA whereas the GF is a set of global supporting functions. Each plane may represent the behavior of the MA in a specific role, so enabling role-based programming, and is composed of local variables (LV), local functions (LF), and an ECA-based automaton (ECAA). This automaton is composed of states and mutually exclusive transitions among states. Transitions are labeled by Event-Condition-Action (E[C]/A) rules, where E is the event name, [C] is a boolean expression based on global and local variables, and A is an atomic action. MAs interact through events that are asynchronously delivered by the MAEE and dispatched, through the Event Dispatcher component, to one or more planes according to the events the planes are able to handle. It is worth noting that the MPSM-based agent behavior programming allows exploiting the benefits deriving from three main paradigms for WSN programming: event-driven programming, state-based programming and mobile agent-based programming.

3.6.4 TinyMAPS

The architecture and the agent programming model of TinyMAPS have been directly derived from MAPS, with proper adaptation to be actually implemented atop the Java-based Sentilla JCreate platform.

Its architecture is depicted in Figure 3.39 and, similarly to MAPS, is based on components interacting through events but offering a more limited set of core-services (agent creation, migration, communication and sensor resource access) to mobile agents. Agent cloning and timer handling are not provided. In particular, the main components are:

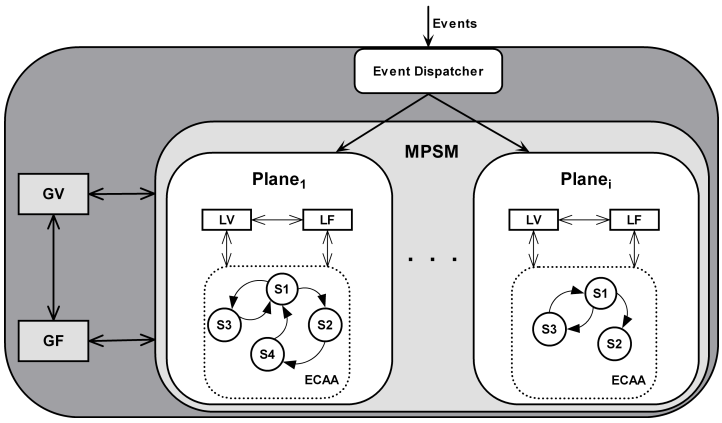


Fig. 3.38. MAPS agent model.

- *Mobile Agent (Agent)*, which is the basic component defined by user. It is designed as a simple class that encloses within the behavior (is possible to define only single-plane agents).
- *Mobile Agent Execution Engine (MAEE)*, which manages the execution of MAs by means of a thread that schedules local or remote events according to a FIFO policy. The MAEE also encapsulates others functions:
  - it uses serialization for sending remote events through the inner component *Mobile Agent Event Sender (MAES)*;
  - it implements a system of naming (*Mobile Agent Naming* component, *MAN*) which keeps the WSN nodes along with the active agents running on them;
  - it interacts with the MAER through the *Mobile Agent Migration Manager (MAMM)* for providing a mechanism of migration for mobile agents.
- *Mobile Agent Event Receiver (MAER)*, which is developed as an independent thread that waits for receiving events from remote MAs. After event reception, it delegates the delivering of event to the MAEE.
- *Resource Manager (RM)*, which allows accessing to the resources of the Sentilla node, i.e. a 3-axial accelerometer and LEDs.

The TinyMAPS mobile agents are defined by following the same multi-plane state machine (MPSM) model previously discussed and depicted in Figure 3.38.

3.6.5 A comparison among Java-based MAS

In the following section we first describe the main characteristics of the Java-based MAS for WSNs and then comparative results of MAPS with both TinyMAPS and AFME are provided.

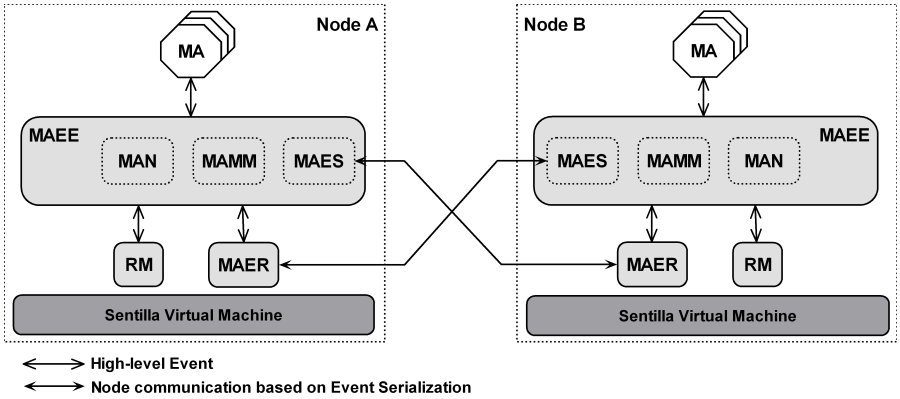


Fig. 3.39. TinyMAPS architecture.

Java-based MASs' characteristics comparison

In Table 3.23, MAPS, TinyMAPS, AFME, and MASPOT are compared with respect to seven characteristics: agent behavior model, intentional agent support, agent behavior definition language, migration type, migration mechanism, agent communication model, and dynamic agent creation.

Table 3.23. Main features offered by the current Java-based MASs for WSNs.

	MAPS	TinyMAPS	AFME [60]	MASPOT [93]
Agent Behavior Model	Finite State Machine	Finite State Machine	Belief/Desire/ Intension	No specific model
Intentional Agent Support	No	No	Yes	No
Agent Behavior Definition Language	Java	Java	Java/AFAPL	Java
Migration Type	Strong (but no code)	Weak	Weak	Weak or Strong
Migration Mechanism	Sun SPOT Isolate	Agent descriptor transmission	Agent descriptor transmission	Sun SPOT Isolate + Suite transfer
Agent Communication Model	Message passing	Message passing	Message passing	Tuple spaces
Runtime Agent Creation	Yes	Yes	No	Yes

Both TinyMAPS and MAPS offer similar services for developing WSN agent-based applications. They use finite state machines (FSMs) to model the agent behavior and directly the Java language to program guards and actions, so no translator and/or interpreter need to be developed and no new language has to be learnt. Moreover, differently from TinyMAPS, MAPS is more power-

ful and fully exploits the Sun SPOT library to provide advanced functionality of communication, migration, sensing/actuation, timing, and flash memory storage. Although AFME is based on the same basic programming language, its agent model is different from a finite state machine, since it employs a more complex BDI-like model, which offers support to intentional agents. In particular, it is centered on perceptors, actuators, modules, and services which are developed in Java but have to be strictly correlated to declarative rules provided for modeling the agent behavior. Both approaches are effective for developing agent-based applications even though MAPS is more straightforward as it relies on a programming style based on state machines widely known by programmers of embedded systems. Differently from the previous systems, MASPOT does not provide any specific model to facilitate developers, which have to design and implement the agents' behavior without the support of a well defined high-level formalism.

For what concerning the migration support, MAPS offers a "limited" strong migration, since the execution state of the agent is transferred during migration along with the agent data state, but no code migration is supported. In particular, the implementation of mobile agents is based on the Isolate components defined by the Sun SPOT library. Each Isolate represents "process-like" unit of computation isolated from other instances of Isolate and their migration mechanism is directly offered by the SPOT SquawkVM through their hibernation and serialization. TinyMAPS, instead, supports migration by simply sending an event that contains agent status information and data, which are encapsulated inside the event. Thus, the agent needs to restart its execution on the remote node. In any case, both MAPS and TinyMAPS suffer from the current limitation of the Sun SPOT and the Sentilla JCreate that, as CLDC-compliant devices do not allow dynamic class loading, so preventing from the possibility to support code migration (i.e. any class required by the agent must be already present at the destination node). Similarly to TinyMAPS, AMFE uses a proprietary agent descriptor to capture and transmit agent data and state. At the contrary, MASPOT supports both strong and weak migration and the type of migration is defined for each agent at creation time and cannot change during the agent life cycle. In particular, along with the migration mechanism based on Isolates, the transmission of Suites (containing a collection of packaged classes and libraries) is employed for migrating the agent code from a central code library situated on the user station (the coordinator computer of the WSN) to a specific Sun SPOT node.

The agent communication model adopted by MAPS, TinyMAPS, and AFME for exchanging information among agents is based on message passing (unicast and broadcast) which is the communication paradigm mostly used in agent-oriented frameworks. The MASPOT inter-agent communication is instead based on the tuple spaces model, similar to the one adopted by Agilla.

The ability to create an agent at runtime could be an important feature for application in which the number of necessary agents cannot be determined "a priori" and simply fixed at compile-time. MAPS, TinyMAPS, and MASPOT

allow for such capability, so providing more flexibility for the creation of dynamic distributed applications, whereas AFME needs agents to be created only in a static way.

3.6.5.1 Performance test comparison between MAPS and TinyMAPS

To evaluate and compare the performance of MAPS and TinyMAPS, two benchmarks have been defined according to [94] for the following mechanisms:

- *Agent communication.* The agent communication time is computed for two agents running onto different nodes and communicating in a client/server fashion (request/reply). Two different request/reply schemes are used: (i) *data Back and Forward (B&F)*, in which both request and reply contain the same amount of data; (ii) *data B*, in which only the reply contains data.
- *Agent migration.* The agent migration time is calculated for agent ping-pong among two single-hop-distant sensor nodes. Migration times are computed by varying the data cargo of the ping-pong agent.

In Figure 3.40 the finite state machines of the agents involved in the two different benchmarks are shown. They are related to both MAPS and TinyMAPS system, since they rely on the same agent modeling formalism.

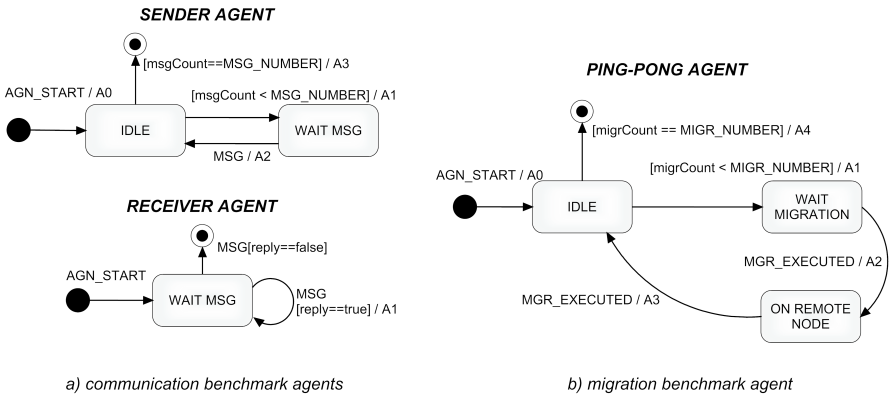


Fig. 3.40. Planes of the agents employed in the benchmarks.

In the *Sender* agent plane, after the agent creation the *AGN_START* event is automatically signaled bringing the agent to the *IDLE* state and executing the *A0* action for some initialization code. From the *IDLE* state, the transition to the *WAIT_MSG* state is immediately triggered whenever the guard

[msgCount < MSG_NUMBER] holds and, consequently, the *A1* action is executed, consisting in sending a message to the *Receiver* agent. The *Sender* then waits until the reply message is received. If so, the *MSG* event is triggered, the action *A2* is executed (the messages exchange time is evaluated and stored), and the plane returns to the *IDLE* state. If the number of messages exchange reaches *MSG_NUMBER*, before the termination of the Sender agent, the operations included in *A3* are performed (i.e. the communication time average is calculated and a last message is sent to the Receiver for its termination). The *Receiver* agent's behavior is very simple. It waits for a message coming from the *Sender* and on the basis of the value of its *reply* parameter, it will send a message reply (*A1*) or terminate itself.

For the agent migration benchmark, a single *Ping-Pong* agent is employed. Upon agent creation and starting, a request for migration is executed (action *A1*) and the plane transits to the *WAIT_MIGRATION* state, waiting for migration completion, which is signaled with the *MGR_EXECUTED* event. After having moved to the remote node, the agent immediately requests for a new migration for coming back to the origin node (action *A2*). Once the agent is came back to the origin node, the elapsed time is stored and a new round-trip migration starst, unless *MIGR_NUMBER* migrations have been completed. Under such a condition, before terminating, the agent computes the migration time average (action *A4*).

The implementation of the agent planes depicted in Figure 3.40 is rather fast, since the basic structure of a generic finite state machine (FSM) is very simple and the main effort for developers is just to insert the code corresponding to the actions of the FSMs, by also making use of the MAPS/TinyMAPS API for accessing to the basic agent management supporting services. An excerpt of the *Sender* agent's plane implementation is shown in Listing 3.2. In particular, the *eventHandler* method is where the FSM and related actions are encode. For more specific technical details on the design and implementation of MAPS/TinyMAPS agents, readers can refer to [95, 96].

In Figure 3.41, the comparison results of the agent communication time, with different message payload, are shown, with MAPS performing better than TinyMAPS. Moreover, as message data payload increases, communication time for MAPS is not affected. The tests have been executed by taking into consideration that the Sentilla JCreate platform imposes a maximum message payload size of 78 bytes.

For what concerning the migration benchmark, Figure 3.42 shows the obtained results. In particular, the migration times are high due to both the slowness of the JVM operations supporting the migration process and the communication time between two nodes. For agents with low data payload TinyMAPS performs better than MAPS; however, when agent data payload is greater than 58 bytes, MAPS migration mechanism starts performing better. Since TinyMAPS relies on messages for transmitting the agent description, the limitation of 78 bytes still holds. At the contrary, MAPS does not have any agent data payload limitations.

Listing 3.2. Excerpt of the Sender agent's plane implementation in MAPS

```
.....
public SenderPlane(Agent agent){
     super(agent);    this.currentState = CREATED;
}
.....
public void eventHandler(Event event){
     try {
        switch(this.currentState){
           case CREATED:
              if (event.getName() == Event.AGN_START){ //action A0
                 while(agents.size() == 0){
                    Thread.sleep(200);
                    agents= this.agent.getRemoteAgentsID();
                 }
                 remoteAgentID= (String)(agents.elementAt(0));
                 this.msgCount= 0;
                 this.currentState = IDLE;
              }
           break;
           case IDLE:
              if(this.msgCount < MSG_NUMBER){ //action A1
                 Event msg = new Event(this.agent.getId(), remoteAgentID,
                                                  Event.MSG, Event.NOW);
                 msg.setParam("msgPayload", msgPayload);
                 msg.setParam("reply", true);
                 this.startTime= System.currentTimeMillis();
                 this.agent.send(this.agent.getId(), remoteAgentID,
                                                      msg, false);
                 this.currentState = WAIT_MSG;
              }else{     //action A3
                 compute&printMean(this.commTime);
                 Event msg = new Event(this.agent.getId(), remoteAgentID,
                                                 Event.MSG, Event.NOW);
                 msg.setParam("reply", false);
                 this.agent.send(this.agent.getId(), remoteAgentID,
                                                      msg, false);
                 this.agent.terminateAgent();
              }
           break;
           case WAIT_MSG:
              if (event.getName() == Event.MSG ) {  //action A2
                 this.commTime[this.msgCount]=
                              System.currentTimeMillis()-this.startTime;
                 this.msgCount++;
                 this.currentState = IDLE;
              }
        }
     }
     catch(Exception e){LedsManager.error(); e.printStackTrace();}
   }
....
```

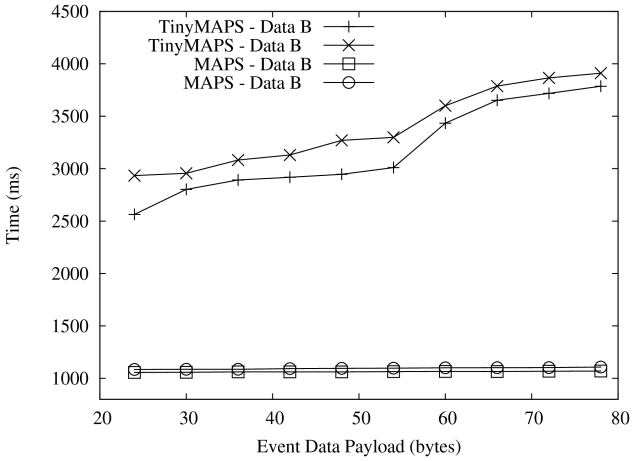


Fig. 3.41. MAPS vs. TinyMAPS: Agent communication time comparison.

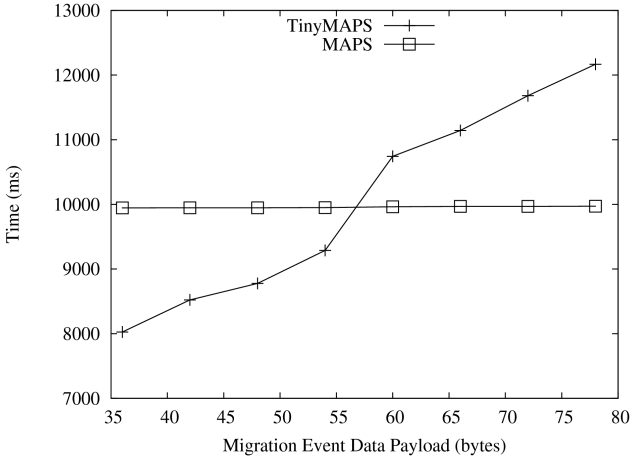


Fig. 3.42. MAPS vs. TinyMAPS: Agent migration time comparison.

3.6.5.2 Performance test comparison between MAPS and AFME

The same benchmarks discussed in Section 3.6.5.1 have been performed for AFME, and the obtained communication/migration performance results are compared with MAPS.

Differently from MAPS and TinyMAPS, AFME agents' behavior is defined through AFAPL declarative rules. In Listing 3.3, the rules of the two agents defined for testing the agent communication time are reported and described.

The first rule of the *Sender* agent checks if less than *MSG_NUMBER* messages have been sent. When the *numMsgSent* belief is adopted, it returns the

Listing 3.3. Sender and Receiver agents' rules in AFME

```
Sender Agent rules:
1. numMsgSent(?msgCount), #?msgCount<MSG_NUMBER > storeStartTime,
                          inform(agentID(receiverAgent),
                            addresses("radiogram://"+receiverNodeAddr));

2. message(inform, sender(receiverAgent, address(?addr))) >
                                compute&storeCommTime, increaseMsgCount;

3. numMsgSent(?msgCount), #?msgCount==MSG_NUMBER > compute&printTimeAverage;

Receiver Agent rule:
1. message(inform, sender(senderAgent, address(?addr))) >
     inform(agentID(senderAgent), addresses("radiogram://"+senserNodeAddr));
```

number of messages sent into the *?nSamples* variable, whose value is tested for confirming that a new message has to be sent. If so, the starting time is also acquired (it is in charge of the *storeStartTime* actuator). The second rule computes and stores the elapsed time for the communication upon the reception of the reply message coming from the *receiverAgent* (see the *message* belief), whereas the messages count is incremented (*compute&storeCommTime* and *increaseMsgCount* are the two actuators in charge of performing such operations). Finally, the third rule fires when the number of messages sent so far is equal to *MSG_NUMBER*, so that the final communication time average can be computed and displayed (i.e. the *compute&printTimeAverage* actuators is executed).

The *Receiver* agent has one rule, which simply consists in sending a message reply whenever a message coming from the *Sender* agent is received.

For what concerning the migration benchmark performed, the needed rules for the correct execution of the *Ping-Pong* agent are shown in Listing 3.4:

The rules defined above are much more complicated with respect to the ones previously defined for the agent communication benchmark, and also much more difficult to read and understand if compared to the simple and clear finite state machine formalism adopted by MAPS/TinyMAPS and depicted in Figure 3.40. In particular, two set of rules are needed for a correct execution of the *Ping-Pong* agent: the rules associated to the agent and representing its running behavior, and a startup rule which is necessary for creating a set of beliefs (*destAddr*, *time*, and *ieeeAddr* along with related values) after the agent creation, but before the agent start, and representing a kind of knowledge initialization. In particular, the *ieeeAddr* belief represents the node address on which the agent is currently running, whereas the *destAddr* belief represents the destination node to which the agent has to migrate.

Upon the agent start, since the aforementioned starting beliefs hold, the first rule fires and the *migrate* actuator is performed for requesting to the AFME middleware the migration of the agent to the destination node, whose address has been previously stored in the *?destaddr* variable by the startup

Listing 3.4. Ping-Pong agent's rules in AFME

```
Startup rule:

1. always(ieeeAddr(com.sun.spot.peripheral.Spot.getInstance()
                   .getRadioPolicyManager().getIEEEAddress()+":45")),
   always(destAddr("0014.4F01.0000.07DB:46")),
   always(init);

Ping-Pong Agent rules:

1. init, destAddr(?destaddr), ieeeAddr(?addr) >
    par(
     migrate(?destaddr,null), retractBelief(always(destAddr(?destaddr))),
     retractBelief(always(ieeeAddr(?addr))), retractBelief(always(init)),
                              adoptBelief(always(couple(?time,?addr))),
                                        adoptBelief(always(migrated)),
                              adoptBelief(always(ieeeAddr(?destaddr))),
                                  adoptBelief(always(destAddr(?addr))),
                                                      time(?time)
    );

2. migrated, destAddr(?destaddr), ieeeAddr(?addr) >
    par(
     migrate(?destaddr,null), retractBelief(always(destAddr(?destaddr))),
                               retractBelief(always(ieeeAddr(?addr))),
        retractBelief(always(migrated)), adoptBelief(always(terminated)),
                              adoptBelief(always(ieeeAddr(?destaddr))),
                                  adoptBelief(always(destAddr(?addr)))
    );

3. terminated, couple(?time, ?addr) > par(printTime(?time, ?addr),
                                  retractBelief(always(terminated)));
```

rule. At the same time (the *par* keyword indicate the parallel execution of commitments/actuators), the *ieeeAddr* and *destAddr* need to be retracted and readopted by swapping their related values. Moreover, the time of starting migration is stored and the *migrated* belief is generated so that the agent, on resume, knows that the migration is completed. The second rule is rather similar to the previous one and is in charge of performing the returning migration to the origin node. Once the agent terminates its ping-pong trip, the third rule fires (i.e. the *couple* and the *terminated* beliefs hold) and the elapsed time is finally computed and displayed.

Along with the AFAPL rules, AFME requires also the implementation of proper Java classes, each of which is related to a specific belief and actuator and represents the actual code that is executed on the Sun SPOT nodes. For more specific technical details on the design and implementation of AFME agents, readers can refer to [95, 96].

The results obtained by the two performed benchmarks are described in the following.

Comparison results for the communication time are shown in Figure 3.43. For messages with light data payload, AFME performs better than MAPS;

however, when the message data payload overtakes 700 bytes, MAPS starts performing better in the case *data BF*.

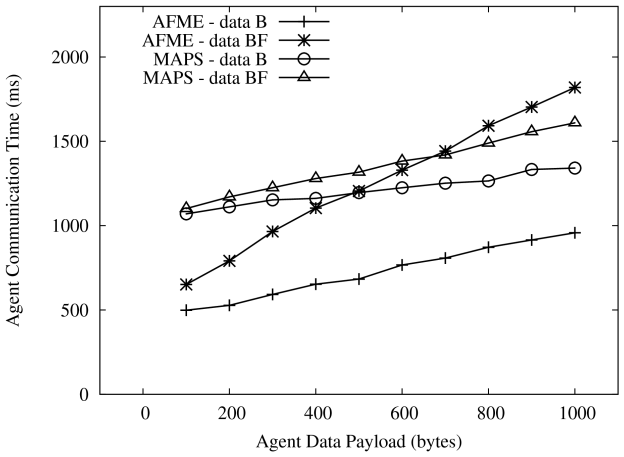


Fig. 3.43. MAPS vs. AFME: Agent communication time comparison.

Comparison results for the migration times are shown in Figure 3.44. AFME retains a higher performance migration mechanism, as it is not based on the heavy isolate hibernation/serialization mechanisms of the SquawkVM.

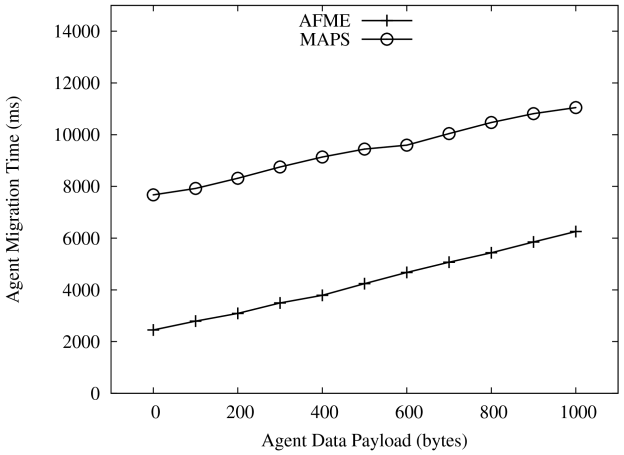


Fig. 3.44. MAPS vs. AFME: Agent migration time comparison.

3.6.6 A case study: agent-based human activity monitoring

In this section we present an agent-oriented signal processing in-node environment specialized for real-time human activity monitoring based on WBSNs. In particular, it is able to recognize postures (e.g. lying down, sitting and standing still) and movements (e.g. walking) of assisted livings. The system is designed and implemented with MAPS at the sensor node side and through Java and JADE at the coordinator side.

3.6.6.1 Design and implementation

The architecture of the system, shown in Figure 3.45, is organized into a coordinator and two sensor nodes.

The coordinator side is based on a JADE agent that incorporates two modules of the Java-based SPINE coordinator[64], developed in the context of the SPINE project[11], which are the SPINE Manager and the SPINE Listener. In particular, the SPINE Manager is used by end-user applications (e.g. real-time activity monitoring application) for sending commands to the sensor nodes. Moreover, the SPINE Manager is responsible of capturing low-level messages and events sent from the nodes through the SPINE Listener, which integrates several sensor platform-specific SPINE communication modules (e.g. TinyOS, Z-Stack, etc), to notify registered applications with higher-level events and message content. A SPINE communication module is composed of a send/receive interface and some components that implement such interface according to the specific sensor platform and that formalize the high-level SPINE messages in sensor platform-specific messages. In this work, the SPINE Listener has been enhanced with a new MAPS/Sun SPOT communication module to configure and communicate with MAPS-based sensor nodes. Such module translates high-level SPINE messages formatted according to the SPINE OTA (Over-The-Air) protocol into lower-level MAPS/Sun SPOT messages through its transmitter component and vice versa through its receiver component. The JADE agent coordinator also integrates an application-specific logic for the synchronization of the two sensors. The SPINE-based real-time activity monitoring application was thus completely reused as well as the SPINE Manager, only the SPINE Listener was modified to account for such enhancement.

The sensor node side (see Figure 3.45) is based on two Java Sun SPOTs sensors respectively positioned on the waist and the thigh of the monitored person. In particular, MAPS is resident on the sensor nodes and supports the execution of the WaistSensorAgent and the ThighSensorAgent. WaistSensorAgent and the ThighSensorAgent have the following similar step-wise cyclic behavior:

1. *Sensing* the 3-axial accelerometer sensor according to a given sampling time (ST);

[11] http://spine.deis.unical.it/

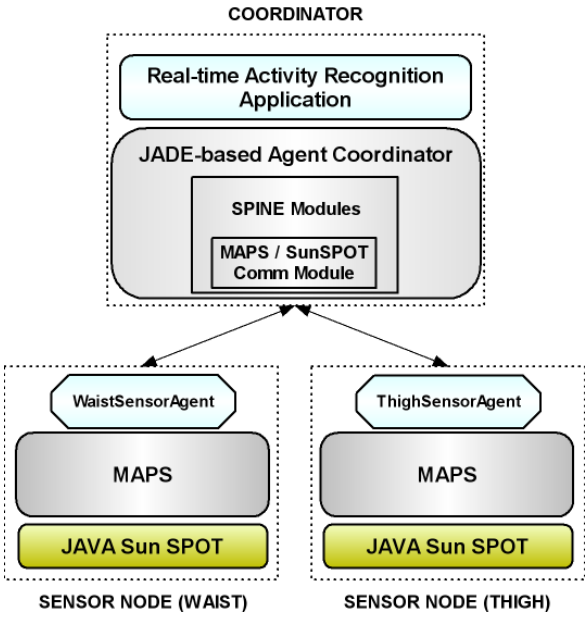


Fig. 3.45. Architecture of the real-time activity monitoring system.

2. *Computation* of specific features on the acquired raw data according to the window (W) and shift (S) parameters. In particular, W is the sample size on which features are computed whereas S is the percentage of sliding on W (usually S is set to 50%);
3. Features *aggregation* and *transmission* to the coordinator;
4. Goto 1.

The agents differ in the specific computed features even though the W and S parameters are equally set. In particular, while the WaistSensorAgent computes the mean values for the accelerometer data sensed on the XYZ axes, the min and max values for data sensed on the X axis, the ThighSensorAgent calculates the min value for data sensed on the X axis.

The interaction diagram depicted in Figure 3.46 shows the interaction among the three agents costituting the real-time system: CoordinatorAgent, WaistSensorAgent and ThighSensorAgent. In particular, the CoordinatorAgent first sends one AGN_START event for each sensor agent to configure them with the sensing parameters (W, S and ST); then, it broadcasts the START event to start the sensing activity of the sensor agents. Sensor agents sends the DATA event to the CoordinatorAgent as soon as features are computed. If the CoordinatorAgent detects that the agents are not synchronized anymore, it sends the RESYNCH event to resynchronize them.

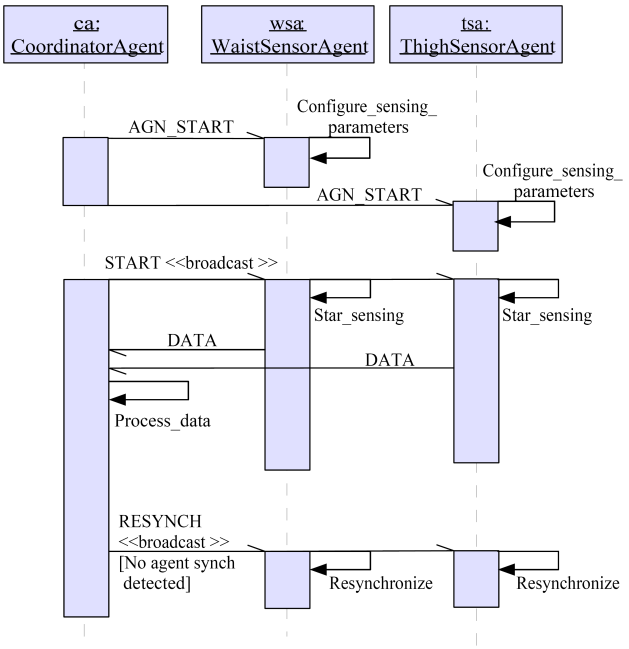


Fig. 3.46. Agents interaction of the real-time activity monitoring system.

The behavior of the WaistSensorAgent is specified through the 1-plane shown in Figure 3.47 and the corresponding code in Listing 3.5 (the behavior of the ThighSensorAgent has the same structure but the computed features are different as discussed above). In particular, after an initialization action (A0) driven by the occurrence of the AGN_START event, the sensing plane goes into the WAIT4SENSING state. The MSG.START event allows starting the sensing process by the execution of action A1, which in particular performs the following steps:

1. sensing parameters (W, S, ST), data acquisition buffers for XYZ channels of the accelerometer sensor (windowX, windowY, windowZ), and data buffers for feature calculation (windowFE4X, windowFE4Y, windowFE4Z) are initialized (see *initSensingParamsAndBuffers* function);
2. the timer is set for timing the data acquisition according to the ST parameter (see *timerSetForSensing* function and in particular the highly precise Sun SPOT timer is used);
3. a data acquisition is requested by submitting the ACC_CURRENT_ALL_AXES event through the sense primitive (see *doSensing* function).

Once the data sample is acquired, the ACC_CURRENT_ALL_AXES event is sent back with the acquired data and the action A2 is executed; in particular:

1. the buffers are circularly filled with the proper values (see *bufferFilling* function);
2. the sampleCounter is incremented and the nextSampleIndex is incremented module W for the next data acquisition;
3. if S samples have been acquired, features are to be calculated, thus sampleCounter is reset, samples in the buffers are copied into the buffers for computing features, calculation of the features is carried out through the *meanMaxMin* function, and the aggregated results are sent to the base station by means of the MSG_TO_BASESTATION event appropriately constructed;
4. the timer is reset;
5. data acquisition is finally requested.

In the ACC_SENSED&FEAT_COMPUTED state the MSG.RESYNCH might be received for resynchronization purposes (see Section 5.2); it brings the sensing plane into the WAIT4SENSING state. The MSG.RESTART brings the sensing plane back into the ACC_SENSED&FEAT_COMPUTED state for (reconfiguring and) continuing the sensing process. The MSG.STOP eventually terminates the sensing process.

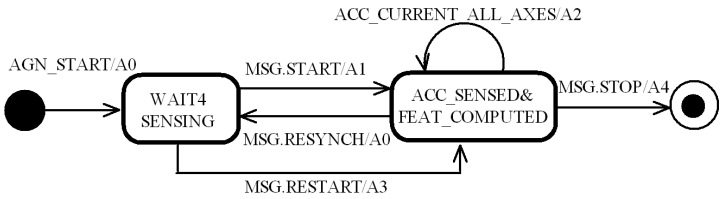


Fig. 3.47. 1-plane behavior of the WaistSensorAgent.

Listing 3.5. Excerpt of the WaistSensorAgent's code in MAPS

```
\textbf{GV}
byte timestamp;
double [] windowX4FE, windowY4FE, windowZ4FE;
String basestationAddress;
\textbf{LV}
int W, S, ST;
byte sampleCounter;
int nextSampleIndex;
IAT91_TC timer;
double [] windowX, windowY, windowZ;
double [] resultsX, resultsY, resultsZ;
\textbf{Actions}
A0: initVars();
A1: initSensingParamsAndBuffers(event);
    timerSetForSensing();
    doSensing();
A2: bufferFilling(event);
    sampleCounter++;
```

```
20      nextSampleIndex=(nextSampleIndex+1)%W;
21      if (sampleCounter==S){
22       sampleCounter==0;
23       copySensingBuffersIntoBuffersForComputingFeatures();
24       computeFeatures();
25       transmitFeaturesComputed();
26      }
27      timerReset();
28      doSensing();
29  A3: timerDisabling();
30      initVars();  A1;
31  A4: timerDisabling();
32  \textbf{LF}
33  initVars():
34    sampleCounter=0; nextSampleIndex=0; agent.timestamp=0;
35    initSensingParamsAndBuffers(Event event):
36    (WaistSensorAgent)agent.basestationAddress=event.getParam(
37                               "BASESTATION_ADDRESS");
38    W=Integer.parseInt(event.getParam("WINDOW_SIZE"));
39    S=Integer.parseInt(event.getParam("SHIFT_SIZE"));
40    ST=Integer.parseInt(event.getParam("SAMPLE_RATE_MS"));
41    windowX = new double[W]; windowY = new double[W]; windowZ= new double[W];
42    (WaistSensorAgent)agent.windowX4FE = new double[W];
43    (WaistSensorAgent)agent.windowY4FE = new double[W];
44    (WaistSensorAgent)agent.windowZ4FE = new double[W];

46  timerSetForSensing():
47    timer = Spot.getInstance().getAT91_TC(0);
48    int cnt = (int)(ST * 1000 / 2.1368);
49    timer.configure(TimerCounterBits.TC_CAPT | TimerCounterBits.TC_CPCTRG |
50                  TimerCounterBits.TC_CLKS_MCK128);
51    timer.setRegC(cnt);
52    timer.enableAndReset(); timerReset();

54  doSensing():
55    Event accel = new Event(agent.getId(),agent.getId(),
56                                  Event.ACC_CURRENT_ALL_AXES,Event.NOW);
57    agent.sense(accel);

59  bufferFilling(Event event):
60    windowX[nextSampleIndex]=Double.parseDouble(
61                          event.getParam(ParamsLabel.ACC_ACCEL_X_VALUE));
62    windowY[nextSampleIndex]=Double.parseDouble(
63                          event.getParam(ParamsLabel.ACC_ACCEL_Y_VALUE));
64    windowZ[nextSampleIndex]=Double.parseDouble(
65                          event.getParam(ParamsLabel.ACC_ACCEL_Z_VALUE));

67  timerReset():
68    timer.enableIrq(TimerCounterBits.TC_CPCS);
69    timer.waitForIrq(); timer.status();

71  timerDisabling():
72    timer.disable(); timer.shutDown();

74  computeFeatures():
75    resultsX = meanMaxMin((WaistSensorAgent)agent.windowX4FE);
76    resultsY = meanMaxMin((WaistSensorAgent)agent.windowY4FE);
77    resultsZ = meanMaxMin((WaistSensorAgent)agent.windowZ4FE);

79  trasmitFeaturesComputed():
80    Event msgToServer = new Event(this.agent.getId(),
81        Constants.MSG_TO_BASESTATION, Event.MSG_TO_BASESTATION, Event.NOW);
82    msgToServer.setParam(ParamsLabel.AGT_BASESTATION_ADDRESS,
83                      (WaistSensorAgent)agent.basestationAddress);
84    msgToServer.setParam("MeanX","" + resultsX[0]);
85    msgToServer.setParam("MeanY","" + resultsY[0]);
86    msgToServer.setParam("MeanZ","" + resultsZ[0]);
```

```
87    msgToServer.setParam("MaxY", "" + resultsY[1]);
88    msgToServer.setParam("MinY", "" + resultsX[2]);
89    (WaistSensorAgent)agent.timestamp=(
90                        (WaistSensorAgent)agent.timestamp+1)%128;
91    msgToServer.setParam("Timestamp", "" +
92                        (WaistSensorAgent)agent.timestamp);
93    agent.send(agent.getId(), Constants.MSG_TO_BASESTATION,
94                                        msgToServer, false);
95  double [] meanMaxMin(double []): //omissis
```

3.6.6.2 Recognition accuracy

The activity monitoring system integrates a classifier based on the K-Nearest Neighbor algorithm [97] that is capable of recognizing postures and movements defined in a training phase. The classifier was setup through a training phase and tested considering the following parameter setting: ST=100ms, W=20 (S=10), P=25%. Accordingly, the features (Min, Max and Mean) are computed on 20 sampled data every new 10 samples acquired by the sensors. The training phase used a KNN-based classifier parameterized with K=1 and the Manhattan distance which performs quite well as classes (lying down, sitting, standing still and walking) are rather separate and scarcely affected by noise. The test phase is carried out by considering the pre-defined sequence of postures/movements represented by the state machine reported in Figure 3.48. Accordingly, the obtained classification accuracy results are reported in Figure 3.49. As can be noted after a transitory period of 5 s from one state to another, all the postures/movements are recognized with an accuracy of 100%. The state transitions more difficult to recognize are STA→SIT, WLK→STA, and SIT→LYG, whereas the transition STA→WLK is recognized as soon as it occurs. The obtained results are good and encouraging if compared with other works in the literature which use more than two sensors on the human body to recognize activities [98].

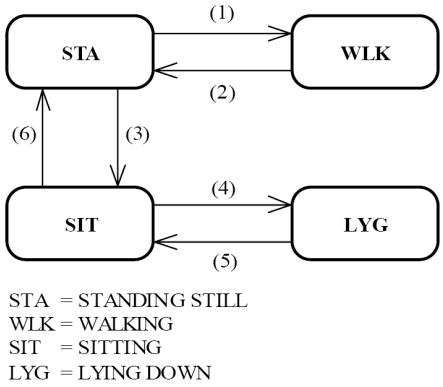


Fig. 3.48. State machine of the pre-defined sequence of postures/movements.

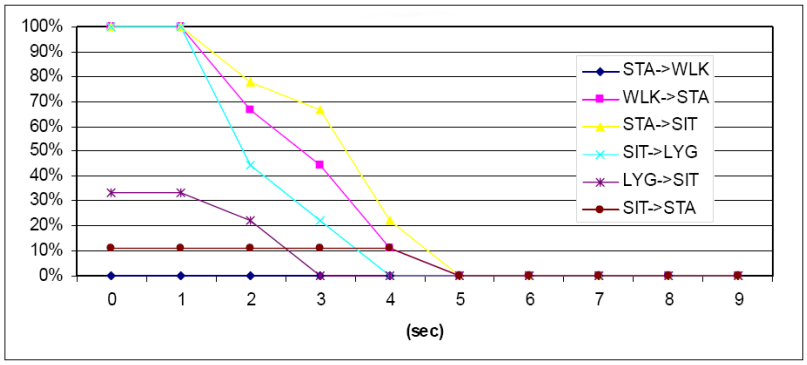


Fig. 3.49. Percentage of mismatches vs. transitory time computed with ST=100 ms, W=20, P=25%.

4

Networking-level protocols

Due to the inherent distributed nature of a WSN, along with proper programming tools for the development of applications with autonomic capabilities, the underlying communication protocols should also provide some kind of self-adaptive property to better support the global behavior of the applications to the network status or changing network conditions as well as to better manage the sensor node resources. And these protocols should possibly show very little computational and communication overhead.

At this purpose, this chapter presents specific networking-level protocols for WSNs having self-adaptive capabilities.

In particular, an enhancement of the AODV routing protocol, called CG-AODV, and a novel contention-based MAC protocol, called QL- MAC, are described in details. CG-AODV adopts a node concentration-driven gossiping approach for limiting the flooding of control packets and improving the network performance in terms of packet delivery ratio and path discovery delay, whereas QL-MAC, based on a Q-learning approach, aims to find an efficient wake-up strategy to reduce energy consumption on the basis of the actual network load of the neighborhood.

For both protocols, simulation results are shown to validate their ability to adapt to the network status and guarantee better network performance with respect to other standard protocols.

4.1 CG-AODV: node concentration-driven gossiping routing protocol

Since many of these applications require sensor nodes to be deployed over a certain geographical area, the scarce wireless communication range implies the use of a multi-hop network organization with no fixed infrastructure. Thus, a routing protocol is necessary to support the high-level distributed applications by ensuring that sensor nodes correctly communicate and cooperate with each others.

Differently from traditional distributed systems, the challenges of routing over a WSN derive from the limited intrinsic characteristics of these networks such as energy and communication bandwidth constraints, scarce node capabilities, absence of IP-based addressing scheme. Moreover, the routing process is also influenced by several other factors that strictly depend on the specific application context and the environmental conditions: node deployment, data delivery models, node/link heterogeneity, connectivity, and coverage. Routing protocols for WSNs have been well studied by the research community. A quite extensive number of routing techniques proposed in the literature as well as their classification can be found in the surveys [99, 100, 101, 102].

The present work does not aim at defining a completely new routing protocol. It is intended to provide an enhancement of a well known reactive protocol on the basis of the results obtained by analyzing how the control traffic impacts the network performance with respect to a new defined quantity, the "node concentration".

In particular, we provides a two-fold contribution. First, we introduce the concept of "node concentration" which considers the average number of neighbors connected to each node, given a specific transmission power (or transmission range). We are interested in investigating how the control traffic generated by a reactive routing protocol impacts the network performance with respect to different values of node concentration. Specifically, we have chosen AODV (Ad-hoc On-demand Distance Vector) [103, 104] as the reference protocol for our analysis, because this is one of the most popular reactive routing protocols and it has been widely adopted by the wireless network research communities. As a second contribution, we propose an enhancement of AODV, by introducing a "node concentration-driven gossiping" approach for limiting the flooding of the route request packets, which we show to be the main reason affecting the network performance. All the experimental analysis have been carried out by means of Castalia/OMNET++[1] simulation platform.

4.1.1 Related work

4.1.1.1 Node density

Node density in WSNs has been investigated in several research works. In [105], the authors analyze the lifetime problem of a WSN from the node density point of view, by considering a periodically data delivery scenario to a sink node. Based on such analysis, a sensor's deployment method aiming at increasing the node density near the sink, is proposed, so that the lifetime of the network is maximized. Another approach for incrementing the lifetime of the network is proposed in [106], where a neighborhood-aware density control is described to reduce the undesired effect of unnecessary overhearing along routing paths. In [107] a new energy distribution model is proposed

[1] http://castalia.research.nicta.com.au

and the relation between wireless sensor network lifetime and wireless sensor node number for both regular and random deployments is investigated. Coverage issues are instead analyzed in [108], where a new notion of "information coverage" is presented. Moreover, the density requirements for complete information coverage of a field are analyzed and simulated for a random sensor deployment. Other works proposing density control approaches for increasing the network lifetime or for coverage purposes can be found in [109, 110, 111].

Although node density in WSNs is a network characteristic that has been investigated in the literature, such works mainly focus on network lifetime or coverage issues, rather than on network performance metrics like the ones more related to the routing problem. Usually, node density is defined as the number of sensors per unit area. By contrast, we consider the average number of neighbors connected to each node, given a specific transmission power (or transmission range).

4.1.1.2 Routing protocols

In recent years, several routing protocols have been specifically proposed for WSNs [99, 100, 101, 102], whereas some others, originally conceived for MANETs, have been tested and adopted in the WSN domain showing good performance results [112, 113].

As we discuss in Section 4.1.3.1, we are interested in reactive routing protocols and specifically in AODV (Ad-hoc On-demand Distance Vector) [103, 104]. Since it has been conceived for MANETs, most of the research works are focused on the study of AODV on such networks, whereas a few works have been devoted to WSNs. Specifically, performance evaluations of AODV on WSNs can be found in [114, 115, 116]. Moreover, improvements of the original AODV protocol have been proposed in [117, 118, 119].

Our study looks specifically at a suitable method to analyse AODV in WSNs and proposes improvements mainly in the path-discovery process. We are also interested in networks featuring node stability, which is a deployment characteristic required by many real-world applications focused on environment monitoring. Moreover, no performance analysis of AODV has been conducted with respect to "node concentration" or "network density" and, to the best of our knowledge, an AODV enhanced with a gossiping technique applied to the WSN domain has not been proposed yet.

4.1.1.3 A brief overview of AODV

AODV belongs to the reactive on-demand routing protocol family [120]. It is based on three different mechanisms: route discovery, route maintenance, and route revocation. When a certain node needs to send a packet to a specific destination, and no valid routing information is already available, a route discovery procedure is initiated by first broadcasting a Route Request (RREQ)

control packet. Such a packet propagates in a flooding way throughout the network. At the same time, the source node sets a timer interval and waits for some responses in the form of Route Reply (RREP) packets. The RREP packet is first generated by the destination node as soon as it receives the RREQ. The RREP packet is in charge of creating a reverse path from the destination to the source node: any intermediate node that receives a RREP appends its own node id information to the reverse path stored in the packet and forwards it to the node from which the RREQ was initially received. In such a way, as soon as the source node originating the discovery phase receives a RREP packet, it is able to forward the data packets to the destination, thanks to the route information stored in the RREP.

Once a route has been established, all nodes that are part of it periodically exchange each others the *Hello* packets for connectivity management (route maintenance phase). If a node does not receive a packet within a specific time interval, it means that the direct link has broken up. As a consequence, a Route Error (RRER) is propagated to the source node so to inform it that the path is no longer available for that specific destination and thus revoking the routing information about the path. Then, if the source node still needs to send further packets, it has to restart a new route discovery phase in order to build a new path.

One of the main problem with the original AODV formulation is that RREQ packets are broadcast throughout the network through flooding, thus wasting substantial bandwidth and energy resources and reducing the network lifetime. Moreover, if a neighborhood is highly dense, all these RREQ exchanges may cause several collisions due to the sharing of the radio channel. Consequently, as a result of frequent necessary retransmissions, network overhead and energy consumption on the nodes increase whereas the whole network performance degrades.

4.1.2 Node concentration

Instead of using the classical definition of node density, which needs to refer to some absolute spatial information, we exploit the local concept of node concentration, K_i, which, with respect to a certain node i, is defined as:

$$K_i = N_i / R_i \tag{4.1}$$

where N_i is the number of neighboring nodes at a 1-hop distance from node i, and R_i represents the transmission range of node i.

The node concentration somehow includes useful information for understanding how a network, or a part of it, behaves with respect to a certain communication pattern among sensor nodes. In fact, if we consider a specific zone of the network, some important communication issues that may affect the quality of service, such as high traffic load, collisions, and congestion, are mostly influenced by the concentration of a neighborhood (and related nodes'

interaction) in that zone rather than the global geographical (or topological) density measured over the same spatial area (which may also include areas without any node). A neighborhood is defined as the subset of nodes able to directly communicate with each others and then sharing the same wireless channel, which is a very limited resource. Thus, node concentration has more meaningful information than the density of a network (or a part of it) intended as the number of nodes per square meter.

Moreover, the node concentration can be locally calculated by each node, contrary to network density, which requires some extra information about the spatial distribution of the nodes (to get such information is not trivial, unless every node is equipped with a GPS receiver). Then, it does not only represent a feature characterizing the network, but also can be a useful information thanks to which each single node may be able to better manage the shared channel resources, by means of specific policies.

4.1.3 Reference scenario and simulation setup

In the following subsections, we first discuss about the WSN scenario we want to focus on and consequently the reasons for having chosen AODV as a reference routing protocol. Afterward, simulation setup is described in details.

4.1.3.1 Reference scenario

The development of a routing protocol is in general based on the application needs and/or the architecture of the network.

Although the existing routing protocols for WSNs may be classified under different points of view [99, 101], most of them are conceived for gathering and forwarding sensor data to a sink node. In fact, differently from traditional networks, WSNs are usually employed in data-collection applications that exhibit an asymmetric traffic patter: sensor nodes persistently send acquired data to a data-collector node (base station or sink) which only occasionally may send back some kind of control messages. In such a context, the network efficiency may benefit from the use of a subset of nodes having the function of relay/aggregator and in charge of collecting and possibly processing data before forwarding it to the base station. Such specialized nodes, usually called cluster heads, oversee and coordinate a specific group of nodes (a cluster) so that the network exhibits a hierarchical organization.

We are not interested in data-collection oriented routing protocols but in protocols enabling point-to-point communication among pair of nodes, useful for building a higher communication layer involving only the subset of cluster-head nodes. One of the possible classifications for point-to-point routing protocols divides them into proactive protocols (also known as table-driven protocols) and reactive ones (also known as source-initiated or on-demand protocols).

Research on routing protocols shows that reactive routing is generally preferred to proactive solutions in different aspects, such as network lifetime, self-organizing network model and the load of the network [121, 122]. It is also well-known that under frequent network topology changes, reactive protocols may suffer from large volume of messaging overhead due to the many necessary route discoveries. But, differently from the MANETs (Mobile Ad-hoc Networks), where routing information may change frequently due to node mobility, in most application scenarios the nodes constituting a WSN are mostly stationary after their deployment which results in predictable and non-frequent topological changes. In such cases, a proactive protocol would represent the most adequate solution for packets routing, as it does not need continuous routing information updates; also path discovery requests are not so frequent as long as routes caching is adopted. However, even with stationary nodes, reactive protocols may be equally subject to some kind of dynamism in the selection of node-pair for communication. It is the case in which clustering is not a fixed process and the cluster-heads election dynamically changes over the time to satisfy applications' requirements. Among the existing point-to-point reactive protocols, we have chosen AODV (Ad-hoc On-demand Distance Vector) [103, 104] as the reference protocol for our analysis, because this is one of the most popular reactive routing protocols and it has been widely adopted by the wireless network research communities.

Moreover, since sensor nodes may need to be densely deployed over a certain area, a reactive protocol may be influenced by the network density. Network density is usually measured as the number of nodes per unit area, and sometimes it is also referred to as just the number of nodes constituting the network over a certain deployment area. However, such a measure in not always sufficient to properly characterize a network, because the actual transmission range of the nodes is not explicitly taken into consideration. In fact, collisions and packet overheads in a specific zone of the network are mostly influenced by the concentration of a neighborhood (i.e. the subset of nodes able to directly communicate with each others) rather than the geographical density of the network intended as the number of nodes per square meter.

4.1.3.2 Simulation setup

All the set of simulations we will discuss in Section 4.1.4 and 4.1.6 have been carried out on the popular Castalia/OMNET++ platform. In particular, as summarized in Table 4.1, the simulations are performed on two scenarios having most of the parameters in common.

First of all, both a regular grid topology and a random one have been employed. A grid topology (the nodes are uniformly placed over the simulation area) is suitable for showing how the network performance is influenced if a homogeneous node concentration all over the network is considered. In both cases, all nodes are statically deployed, i.e. do not move over the simulation time. As for the low-level details, all nodes have been configured with the

same radio equipment, the CC2420, having the same parameters, such as a transmission power level of 0dBm, which roughly corresponds to a 46 meters transmission range. The MAC layer simply adopts a CSMA/CA channel access method, without relying on the RTS/CTS (Request To Send / Clear To Send) packet exchanges.

Concerning the differences, in "Scenario 1" the squared simulation area has been maintained fixed (i.e. 200 meters per side), whereas the number of nodes has been varied from 25 (grid of 5x5) to 144 (grid of 12x12). In "Scenario 2", a fixed amount of nodes (i.e. 100) have been deployed on different areas ranging from 25x25 meters to 400x400 meters. As a consequence of such deployments, each specific configuration corresponds to a certain value of the node concentration.

In the following, the communication pattern is explained. The aim of these simulations is to analyse the behavior of AODV on a network organized in a hierarchical way, where only few nodes (representing the upper layer ones, i.e. cluster heads) communicate with each others. More specifically, the whole network has been divided into four to twelve sub-areas (representing the clusters) and for each of them a node (the cluster-head) is chosen as the one in charge of sending/receiving the data packets. Since we are not interested in the clustering phase (we assume that the clustered network has already been formed), the remaining nodes are only in charge of forwarding packets among cluster heads. On a variable time interval, so that the rate is roughly 2 or 8 pkt/sec, each of the cluster-heads randomly chooses one of the remaining ones and sets it as the destination node for the next data packet to be delivered. Then, the mechanisms inside the AODV protocol take place to establish (if not already done) the proper route and, finally, the data packet is properly sent throughout the network.

Although such scenarios may appear simple both in the network and in the communication pattern configuration, this is sufficient to show how the routing performance is significantly affected by an increase in node concentration, due to the increasing chance of collisions among routing control packets (RREQ, RREP) and data packets.

It is worth noting that, for each specific configuration the performance metrics have always been averaged over 50 independent runs.

4.1.4 Analysis of AODV with respect to "node concentration"

As a first contribution, in the following section, we present some simulation results carried out on the standard AODV protocol. In particular, we are interested in analyzing the AODV behavior under different node density conditions of the network. To this extent, we are not interested in measuring the network density simply as the number of nodes per unit area, but we use the "node concentration".

In order to understand the impact of the control traffic on network performance, the Packet Delivery Ratio (PDR) represents an effective performance

Table 4.1. Parameters used in the simulations.

	Scenario 1	**Scenario 2**
# of nodes:	25, 36, ..., 144	100
Sim. Area [mxm]:	200x200	25x25, 50x50, ..., 400x400
Node deployment:	"Regular grid" and "Random" topologies, with static nodes	
Communication pattern:	4/8/12 cluster-heads communicating with each others	
Packet rate:	$\approx$ 2 and 8 pkt/sec	
Data packet payload:	32 bytes	
MAC Layer:	CSMA/CA-based	
Radio device:	CC2420	
Transmission power:	0dBm	
Simulation runs:	50 for each specific configuration	

metric capable of highlighting the efficiency of the routing protocol under specific conditions.

All the simulations graphs reported in the following section have been carried out on networks deployed in regular grid topologies, divided into four clusters and with an application traffic of 2 pkt/sec. In particular, as shown in Figure 4.1, two different plots have been obtained. The Castalia simulator offers the user the opportunity to opt for different interference/collision models. In a first case, the simpler model is used, which means that collisions are not handled and two overlapping transmissions at the same time do not interfere each other. Thus, independently from the actual node concentration, the PDR remains practically the same. But, in a real environment, collisions play an important role in the network performance. This is demonstrated by setting the simulator so to use an additive interference model, through which transmissions from other nodes are calculated as interference by linearly adding their effect at the receiver. As a consequence, we obtain a more realistic result, i.e. the PDR decreases with an increase in the node concentration, since collisions are more likely to happen.

It is worth noting that, the plots in Figure 4.1 include results coming from both the simulation scenarios of Table 4.1. This is the reason why several points are overlapping with each other. In fact, configurations with nodes deployment having different densities (nodes per unit area), have actually the same node concentration.

In Figure 4.2, the packet transmission failure rate, under a realistic collision model, is depicted. As expected, failures increase with an increase in node concentration.

4.1.5 CG-AODV: a node concentration-driven gossiping approach

On the basis of the results shown in Section 4.1.4, it is quite clear that the problems affecting AODV are due to the large number of collisions occurring during data packet transmission. Since we have opted for a not so high data

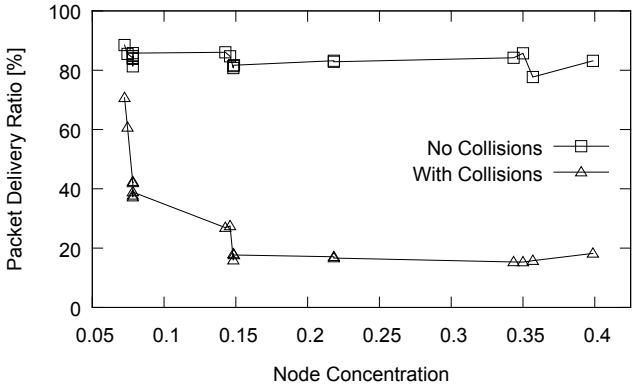


Fig. 4.1. Packet Delivery Ratio on grid topologies and 2 pkt/sec.

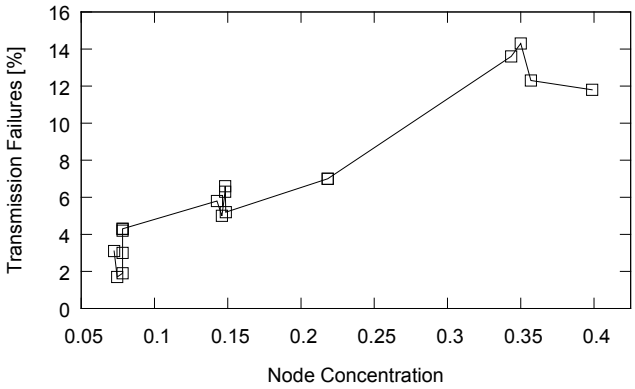


Fig. 4.2. Transmission Failures on grid topologies, 2 pkt/sec, and packet collisions.

packet rate, it is obvious that such collisions are mainly due to the AODV routing control packets and, in particular, to the RREQ ones which are propagated in a flooding way throughout the network.

In the following, we describe an enhancement of AODV (called CG-AODV) which introduces a mechanism to limit flooding, and then reduce the protocol overhead, through the use of a gossip-based filtering/propagation decision. In particular, a *node concentration-driven gossiping* approach is employed so that the gossiping mechanism is not fixed but shows a kind of adaptive behavior on the basis of the nodes' neighborhood concentration.

Similarly to the standard AODV, only the nodes initiating a new route discovery (i.e. the source nodes) are allowed to send the route request to all their neighbors. The concentration-driven gossiping phase takes place on the nodes that receive a RREQ packet from one of their neighbors. The diagram depicted in Figure 4.3 represents the decision process made on each node for

deciding whether forwarding a received RREQ packet or not. First of all, a preliminary check of the packet is done on the basis of a "blacklist". When a RREQ related to the request for a specific path is dropped because of the gossiping mechanism, it is inserted in the blacklist. When a RREQ packet for the same path is received within a certain (relatively short) amount of time, the node immediately drop it, without reprocessing it with the gossiping mechanism.

Differently from other gossiping approach, the probability p adopted for deciding whether discarding a packet is not fixed, but it depends on the node concentration locally measured by each node (which needs to know only about its neighbours). In such a way, the gossiping approach adapts itself on the basis of the local conditions of the network. The larger is the number of neighbors, the greater is the probability p of discarding the RREQ to avoid its flooding propagation.

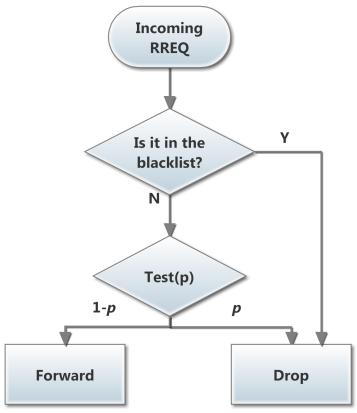


Fig. 4.3. Gossiping mechanism in CG-AODV.

4.1.6 Simulations and results

A set of simulations have been carried out both on regular grid topologies and on random ones. All parameters are the same as the ones previously described in Section 4.1.3.2 and summarized in Table 4.1.

4.1.6.1 Grid topology

In Figure 4.4 and Figure 4.5, the PDR and the Path Discovery Delay (PDD) of CG-AODV compared to the original AODV are shown. In particular, simulation results of AODV with both ideal and realistic collision models are illustrated, whereas CG-AODV has been directly simulated by adopting the

realistic collision model. Moreover, four cluster-heads have been considered and the packet rate set to 2 pkt/sec. As it can be seen, the proposed AODV enhancement guarantees a much greater PDR with respect to the AODV with a realistic collision model and its performance is even very close to the one shown by the theoretical best AODV (i.e. without collisions). As for the PDD, the CG-AODV, in average, requires a little more time to establish a path because of the use of a gossiping RREQ propagation. Anyway, even in the worst case (high node concentration), the resulted delay is no greater than 1.4 seconds.

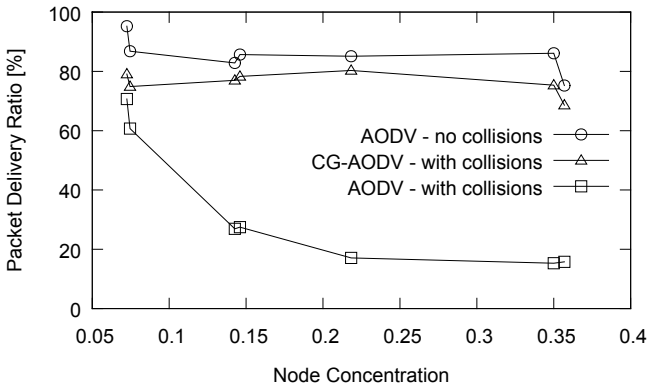


Fig. 4.4. Packet Delivery Ratio comparison on grid topology, 4 cluster-heads, 2 pkt/sec.

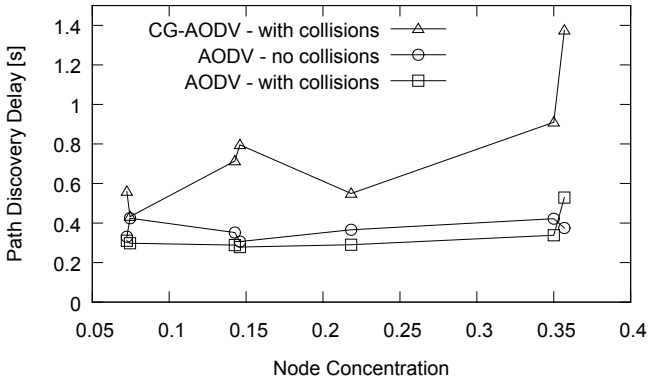


Fig. 4.5. Path Discovery Delay comparison on grid topology, 4 cluster-heads, 2 pkt/sec.

4.1.6.2 Random topology

After having tested CG-AODV on a grid topology so to directly compare the results with the ones coming from the preliminary analysis of the standard AODV (see Section 4.1.4), in the following simulations we consider more generic random topology networks. It is worth noting that, since the network does not have a regular topology, the values for the node concentration have been calculated offline by averaging the node concentration measured on each single node.

In Figure 4.6 and Figure 4.7, simulation results related to PDR and PDD are shown. CG-AODV and the standard AODV have been simulated by taking into account only the realistic collision model, four cluster-heads, and a packet rate of 2 pkt/sec. In this case, the CG-AODV behaves better than AODV by showing higher performances in both network metrics.

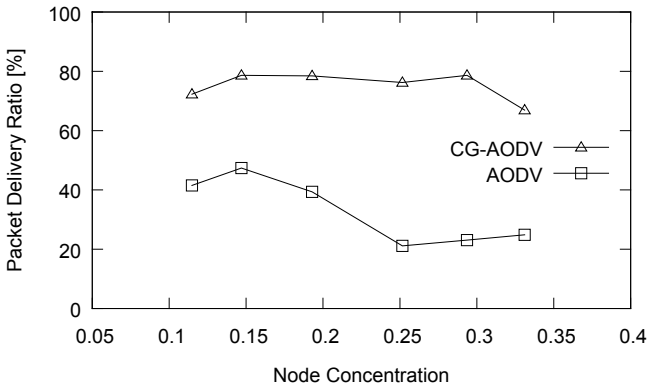


Fig. 4.6. Comparison of Packet Delivery Ratio on random topology, 4 cluster-heads, 2 pkt/sec.

If we compare the above CG-AODV results with the ones related to grid topologies (see Figure 4.4 and Figure 4.5), it can be seen that the proposed protocol has similar performances for PDR on both topologies, whereas the PDD get improved on randomized networks.

All previous simulations have been carried out by maintaining both the number of cluster-heads and the packet rates to a fixed value, i.e. 4 and 2pkt/sec respectively. In the following, further simulation results have been carried out on network scenarios having a greater number of clusters as well as a heavier application packet traffic.

In particular, in figure Figure 4.8 and Figure 4.9 the PDR has been evaluated with 2 or 8 pkt/sec and by varying the number of cluster-heads, i.e. 4, 8 and 12. Both graphs clearly show the benefits provided by CG-AODV which guarantees better performances in all cases, independently from the network

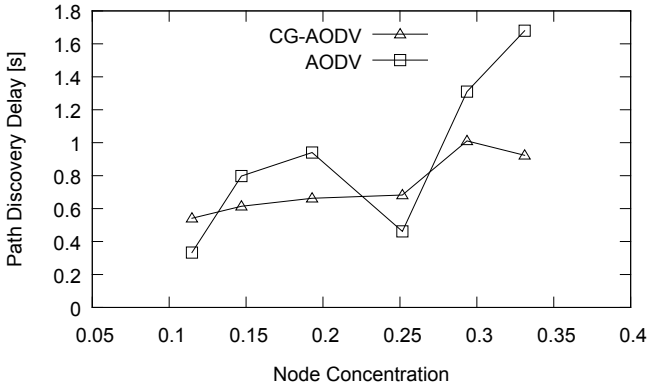


Fig. 4.7. Comparison of Path Discovery Delay on random topology, 4 cluster-heads, 2 pkt/sec.

configuration. In particular, in the standard AODV protocol the PDR never exceed 20% in most of the simulations, thus demonstrating how it suffers in performance under heavy traffic load conditions due to a greater network congestion that increases chance of packet collisions.

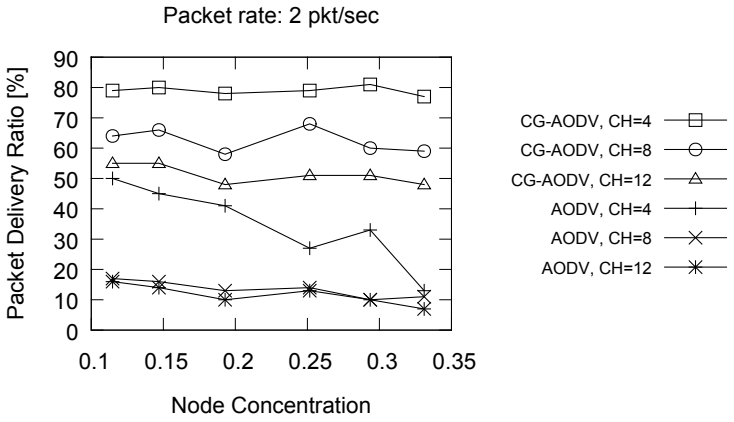


Fig. 4.8. Comparison of Packet Delivery Ratio on random topology, 2 pkt/sec, by varying the number of cluster-heads.

Differently from the PDR results, the comparisons on the PDD metric show that at low packet traffic load both protocols have a similar performance, independently on how many communicating cluster heads are in the network (see Figure 4.10). With a higher packet rate (Figure 4.11), AODV

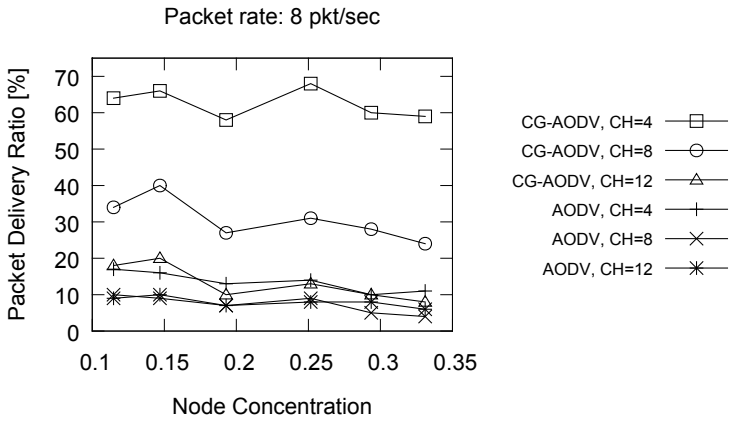


Fig. 4.9. Comparison of Packet Delivery Ratio on random topology, 8 pkt/sec, by varying the number of cluster-heads.

and CG-AODV have similar results when node concentration is low. At higher concentration, the PDD related to AODV shows better results with 4 and 8 cluster heads, but CG-AODV outperforms AODV when the number of cluster heads is increased to 12.

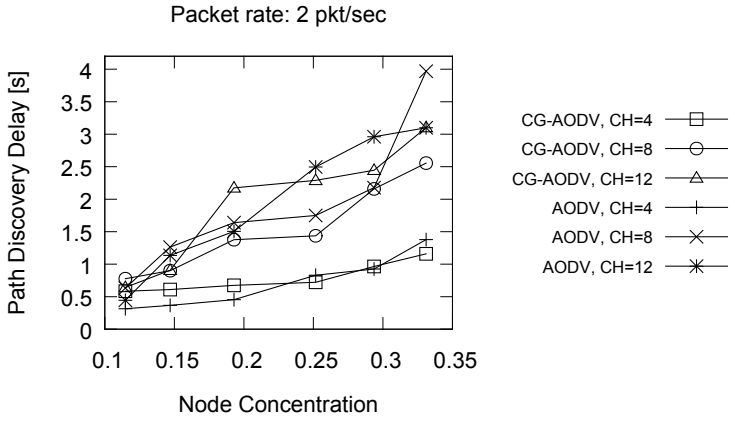


Fig. 4.10. Comparison of Path Discovery Delay on random topology, 2 pkt/sec, by varying the number of cluster-heads.

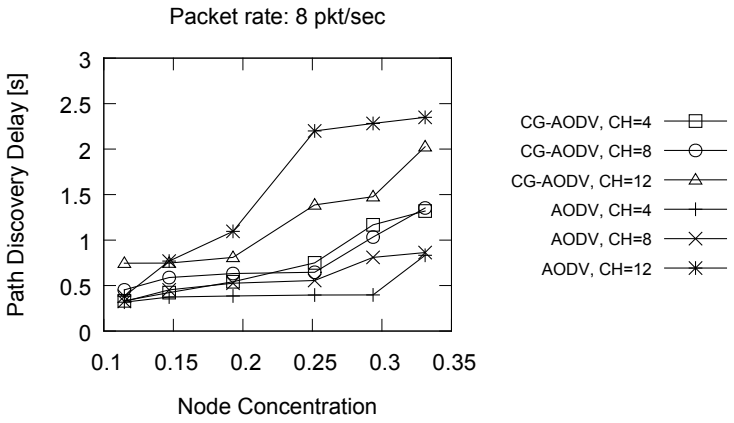


Fig. 4.11. Comparison of Path Discovery Delay on random topology, 8 pkt/sec, by varying the number of cluster-heads.

4.2 QL-MAC: a Q-Learning based MAC for WSN

The aim of the proposed protocol is to allow nodes to infer each other's behaviors in order to adopt a good sleep/active scheduling policy that dynamically learn over the time to better adapts to the network traffic conditions. Specifically, each node, not only takes into consideration its own packet traffic due to the application layer, but also considers its neighborhood's state.

The basic underlying behavior of the QL-MAC is similar to most of other MAC protocols: a simple asynchronous CSMA-CA approach is employed over a frame-based structure. It basically divides the time into discrete time units, the *frames*, which are further divided into smaller time units, the *slots*. Both frame length and slot number are parameters of the algorithm and remain unchanged at execution time.

By means of a Q-Learning based algorithm, each node independently determines an efficient wake-up schedule in order to limit as much as possible the number of slots in which the radio is turned on. Such a non-fixed and adaptive duty-cycle reduces the energy consumption over the time without affecting the other network performances, as shown by the simulations results discussed in Sect. 4.2.4.

4.2.1 Related work

Some of the simplest MAC protocols for wireless networks rely on the time division multiple access (TDMA) [123], which is a "contention-free" approach, where a pre-defined time slot is reserved for each node in each frame. Although such a fixed duty cycling does not suffer from packet collisions, it needs an extremely exact timing in order to avoid critical behaviors.

S-MAC [124] is a contention-based MAC protocol aiming at reducing energy consumption and collisions. It divides time into large frames, and each frame into two time portions (a sleeping phase and an active phase). Compared to the TDMA approach, S-MAC requires much looser synchronization among neighbouring nodes. However, due to a fixed duty cycle it is not capable of adapting to network traffic condition.

The Timeout-MAC (TMAC) protocol [125] is an improvement of S-MAC as it uses an adaptive duty cycle. In particular, by means of a time-out mechanism it detects possible activities in its vicinity. If no activity is detected during the time-out interval, the node goes to sleep for a certain period of time. Such a mechanism occurs every time a communication between two nodes is over. Although T-MAC outperforms S-MAC, its performance degrades under high traffic loads.

In the P-MAC [126] protocol the sleep-wakeup schedules of the sensor nodes are adaptively determined on the basis of a node's own traffic and that of its neighbours. The idle listening periods, which are source of energy wastage, are minimized by means of some kind of matching algorithm among patterns of schedules in the neighbouring.

Other adaptive MAC protocols have been proposed in the literature and few of them employ online machine learning approaches such as reinforcement learning [127, 128] and Q-learning [129].

4.2.2 Reinforcement Learning and Q-Learning

Reinforcement Learning (RL) [130] is a sub-area of machine learning concerned with how an agent take actions so as to maximize some kind of long-term reward. In particular, the agent explores its environment by selecting at each step a specific action and receiving a corresponding reward from the environment. Since the best action is never known a-priori, the agent has to learn from its experience, by means of the execution of a sequence of different actions and deducing what should be the best behavior from the obtained corresponding rewards.

One of the most popular and powerful algorithm based on RL is Q-Learning, which does not need the environment to be modelled and whose actions depend on a so called *Q-function*, which indicates the quality of a specific action at a specific agent's state. Specifically, the Q-values are updated as follows:

$$Q(s_{t+1}, a_t) = Q(s_t, a_t) + \lambda[r_{t+1} + \phi \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \qquad (4.2)$$

where $Q(s_t, a_t)$ is the current value at state s_t, when action a_t is selected.

At some state s_t, the agent selects an action a_t. It finds the maximum possible Q-value in the next state s_{t+1}, given that a_t is taken, and updates

the current Q-value. The discounting factor $0 < \phi < 1$ gives preference either to immediate rewards (if $\phi << 1$) or to rewards in the future (if $\phi >> 0$), whereas the learning rate $0 < \lambda < 1$ is used to tune the speed of learning.

4.2.3 Protocol details

The actions available to each agent/node consist in deciding whether it should stay in active or in sleep mode during each single time slot. Thus, the action space of a node is determined by the number of slots within a frame.

Every node stores a set of Q-value, each of which is coupled to a specific slot within the frame. The Q-value represents an indication of the benefits that a node has when is awake during the related time slot. The Q-value is updated over the time on the basis of some specific events occurring during the same slot at each frame. Moreover, it is also dependent on some state information coming from the node's neighbours.

Specifically, every Q-value related to a specific node i is updated as follows:

$$Q_s^i(f+1) = (1-\lambda)Q_s^i(f) + \lambda R_s^i(f) \tag{4.3}$$

where $Q_s^i(f) \in [0,1]$ is the current Q-value associated to the slot s on the frame f, $Q_s^i(f+1)$ is the updated Q-value, which will be associated to the same slot s but on the next frame, λ is the learning rate and R_s^i is the earned reward. Differently from the update rule shown in Equation 4.2, the future reward is not considered and the discount factor ϕ is set to 0.

In such a decentralized approach, it is important to define a suitable reward function that consider both the condition of the node and the one of its neighborhood. Specifically, the events that the protocol takes into considerations are related to the packet traffic load, so that the reward function calculated on node i and related to a specific slot s is modelled as follow:

$$R_s^i = \alpha\left(\frac{RP - OH}{RP}\right) + \beta S_i + \gamma\left(\frac{\sum_{j=1}^{|N_i|} P_j}{|N_i|}\right) \tag{4.4}$$

where:

- OH is the number of over-heard packets, i.e. the packets received but actually not intended for node i;
- RP is the total amount of packets received by node i during the slot s of frame f. It includes also over-heard packets;
- S_i has a value of $+1$ if node i has at least one packet to broadcast during slot s, 0 otherwise;
- P_j has a value of $+1$ if the neighbouring node j has sent at least one packet to node i during slot s, 0 otherwise;
- N_i is the set of neighbours of node i;
- the constants α, β, and γ weigh the different terms of the function accordingly.

It is worth noting that, at the beginning, all the Q-values on every node are set to 1, meaning that all nodes have their radio transceiver ON on every slot (i.e. for the entire frame). During the learning process, the Q-values changes over the time accordingly to the variation of the reward function. In order to properly set the state for the radio transceiver on the basis of the Q-values, we employ a further parameter T_{ON}, which represents a threshold value:

$$Radio_{[slot\ s]} = \begin{cases} On & \text{if } Q_s^i(f) \geq T_{ON} \\ Off & \text{otherwise} \end{cases}$$

In case the MAC packet exchange takes place always in broadcast mode, so that a node is not able to figure out whether each single received packet is actually destined for itself or not, it is necessary to get some extra information from the upper layers. In particular, our MAC protocol employs a simple cross-layer communication: every received packet are decapsulated and delivered to the network layer, which in turns checks whether the packet is intended for the node. In case the packet is discarded, the network layer signals the MAC protocol about the reception of a overheard packet, and the reward function is updated accordingly to Equation 4.4.

If the radio is turned off at a specific slot but at some point, during the same time window, the node needs to send a packet, we prefer to buffer it and postpone its transmission on the next available slot (i.e. the first one with the radio "on").

The last term of Equation 4.4 is an aggregated information about the state of the node neighborhood and, in particular, it represents the packet traffic activity during a specific time slot. This is the only information exchanged by the protocol. This is fundamental when the node is in sleep mode at a specific slot, so to figure out that it should be better to turn on the radio because of the presence of packets destined for it.

4.2.4 Simulations and evaluation

QL-MAC is simulated and evaluated in Castalia[2]. In the following, the simulation scenarios are first described and then the obtained results are discussed. In particular, the performance evaluation considers two metrics: the average energy consumption of nodes and the Packet Delivery Ratio (PDR). Under these metrics, QL-MAC is compared to two well known MAC protocols for WSNs, SMAC and TMAC, as well as to a simple asynchronous CSMA-CA.

4.2.4.1 Scenario and traffic model

The protocol has been tested on two different scenarios, one with a regular grid topology (nodes have been uniformly placed) and the other with a random one. For both scenarios, the parameters setting summarized in Table 4.2 have

[2] http://castalia.research.nicta.com.au

been adopted. All nodes have been set with the same radio transceiver, the CC2420 with a transmission power level of 0dBm (which allows roughly 46 meters transmission range).

Table 4.2. Parameters used in the simulations.

Scenario Parameter	Value	QL-MAC Parameter	Value
# of nodes:	16	Frame length	1 sec
Sim. Area [mxm]:	100x100	Slot number	4
Radio device:	CC2420	λ	0.05
Transmission power:	0dBm	$alpha$	0.33
Collision model:	real	β	0.33
Packet rate:	$\approx$ 1 pkt/sec	γ	0.33
Data packet payload:	32 bytes	T_{ON}	0.40
Routing protocol	Multipath Rings Routing		
Initial energy	10000 J		

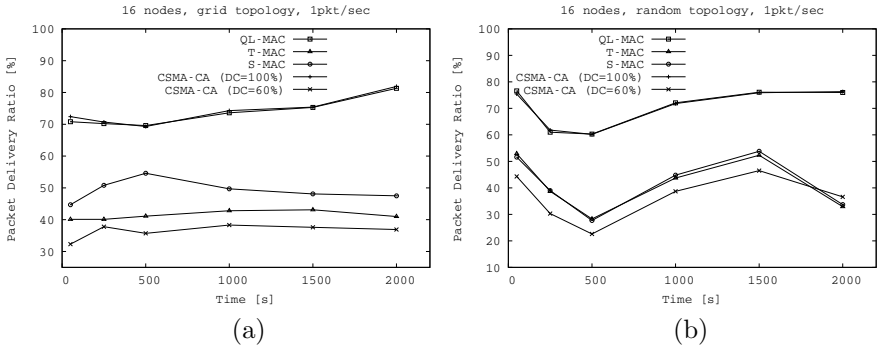
In the following, the communication pattern is detailed. The application we consider in our simulations employ a nodes-to-sink communication pattern, since data-collection applications are one of the most typical use cases of a WSN in real contexts. The sensor data acquired by all the nodes are sent to a sink node centered in the middle of the simulation area. Since the sink is not in the transmission range of every node, a simple multipaths ring routing has been used as a network layer protocol. During an initial setup phase, the sink broadcasts a specific packet with a counter set to 0. Once the packet is received by a node, it sets its own level/ring number to 0, increments the counter and rebroadcasts the packet. This process goes further on until all nodes get their ring level.

After this initial setup phase, every node has a ring number representing the hop distance to the sink. When a node has data to send, it broadcasts a data packet by attaching its ring number. Only the neighbours with a smaller ring number process the packet (i.e. attach its own ring number) and rebroadcast it. This process goes further on until the data packet gets to the sink.

4.2.4.2 Results

In order to understand the impact of the dynamic radio schedule adopted by QL-MAC on the network performance, the PDR has been first analysed, as shown in Figure 4.12. As it can be seen, both on grid and random topology, QL-MAC outperforms all other MAC protocols with the exception of the CDMA-CA with a 100% duty cycle. In this case, both protocols have almost the same performance because of the underlying QL-MAC channel access, which is essentially the same.

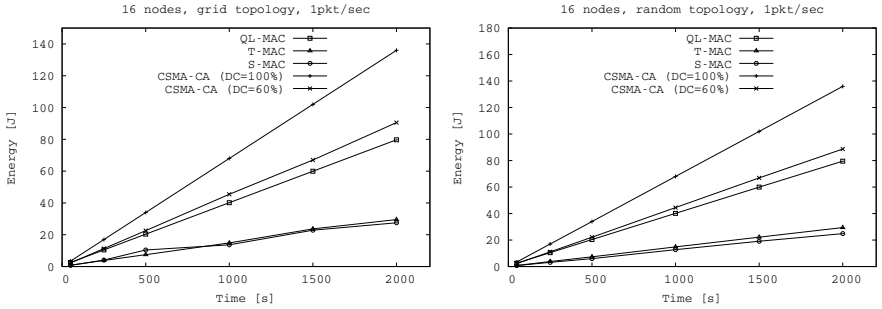
Although both QL-MAC and CSMA-CA (with a 100% duty cycle) share the same PDR performance, their comparison result changes if we consider the



(a) (b)

Fig. 4.12. Packet Delivery Ratio on grid (a) and random (b) topologies.

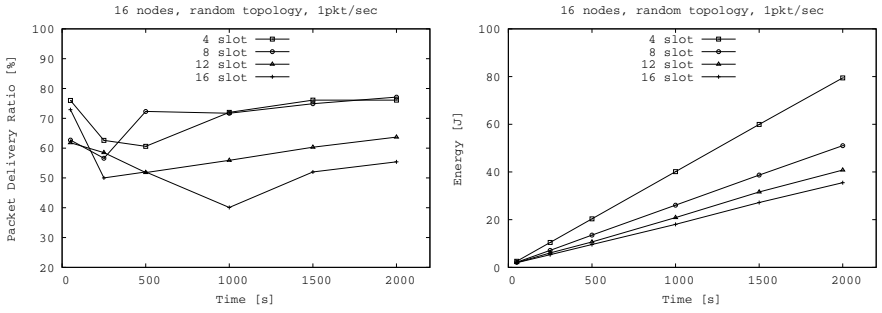
node energy expenditure. In fact, as shown in Fig. 4.13, QL-MAC allows nodes to spend much less energy, as a result of the sleep/wake-up radio schedule. Moreover, it performs better even if compared to a CSMA-CA having a duty cycle of 60%. Both S-MAC and T-MAC show lesser energy consumption but, because of their limited capabilities, they are not able to adapt well to the network traffic pattern.



(a) (b)

Fig. 4.13. The average energy consumption per node on grid (a) and random (b) topologies.

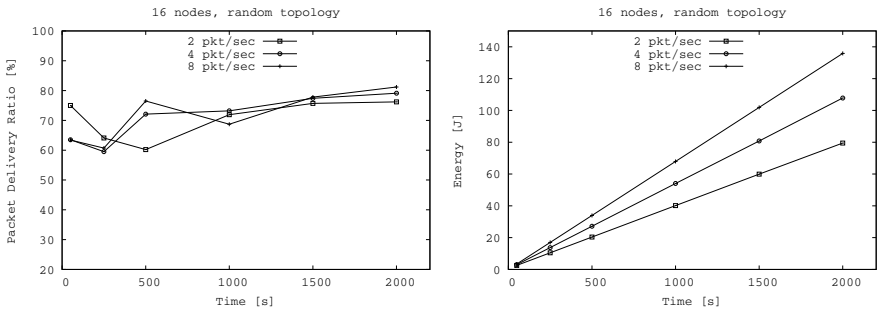
QL-MAC has been also evaluated by varying the number of slots constituting the frame. In Figure 4.14, the simulation results of both PDR and average energy consumption per node are depicted. In general, as the number of slots decreases, QL-MAC shows better performance with respect to the PDR but, as a consequence, the energy spent by node tends to increase. Actually, with the use of 8 slots, the protocol exhibits the better trade-off, i.e. the PDR is similar to the case with 4 slots, but the energy spent is less.



(a) (b)

Fig. 4.14. QL-MAC: Packet Delivery Ratio and average energy consumption per node by varying the slots number.

The results shown in Figure 4.15 are obtained by varying the packet rate of the application layer, from 2 pkt/sec to 8 pkt/sec. As it can be seen, the PDR plots are similar over the time, demonstrating that QL-MAC behaves well under different traffic loads. But, since different amount of packets per time unit are transmitted and delivered, the energy expenditure increases with the increase of the packet rate.



(a) (b)

Fig. 4.15. QL-MAC: Packet Delivery Ratio and average energy consumption per node by varying the application packet rate.

5

Conclusions, Future Directions and Publications

5.1 Conclusions

Throughout this thesis, several contributions have been made to the Wireless Sensor Networks research community.

In particular, this thesis has proposed novel frameworks aiming at supporting rapid development of WSN applications as well as enabling self-management behaviors at runtime, and specific networking-level protocols showing self-adaptive capabilities.

As first contributions, we have proposed the SPINE2 framework, which has been specifically conceived for supporting developers in a rapid and effective prototyping of signal-processing applications for WSNs. By providing an intuitive and straightforward task-based modeling paradigm, it allows to define efficient distributed applications on heterogeneous embedded environments with very few efforts. In order to address the very strict requirements in terms of efficiency and robustness, SPINE2 comes with a node-side middleware carefully designed for achieving high-performance execution of the task-based applications. Moreover, its well-designed underlying architecture is prone to be easily extended with new functionalities and capabilities (i.e. definitions of further tasks or integration of new physical sensor drivers) or even extended to support new sensor platforms through a quick porting procedure, thanks to its platform-independent architecture. The runtime performances of SPINE2 have been evaluated in terms of task execution time and memory usage, so to demonstrate its practical use and efficiency in resource-constrained embedded environments. To further show the feasibility and the effectiveness of using the proposed framework, a real case study in the BSN context has been taken into consideration. In particular, a distributed human action recognition system, relying on wearable sensor nodes, has been developed by means of the SPINE2 task-oriented approach. The resulting application definition and deployment phases have demonstrated how simple and fast is the use of SPINE2 in building complex distributed BSN applications, without which it would be

necessary much more efforts in terms of programming and debugging time. The performance analysis have also shown that such benefits in supporting intensive sensing and data processing applications come with negligible performance penalties.

On the basis of the experience gained in the context of the SPINE2 project[1], we can state that, apart from efficient and robust wearable sensors and low-power standard communication protocols, the need for software abstractions, frameworks, and tools to support an effective development of WSN systems represents one of the main requirements for determining the success of this technology. Furthermore, the availability of specific APIs on the coordinator-side and also a graphical tool for application definition, can improve the productivity in a significant way, since developers do not have to waste time in using low-level programming languages or dealing with hardware management and configuration. Not less important for a framework is its extendibility, which can determine the constant framework evolution and then its practical use over the time. Thus, along with the characteristics of runtime efficiency and programming effectiveness, a framework should have a well-designed inner architecture (at node-side and coordinator-side) suitable for adding new features in an easy way.

To the best of our knowledge, SPINE2 is currently the only framework that encompasses, in a single integrated tool, distinctive characteristics able to fully address the requirements for an effective programming of BSN applications: (1) specific support for in-node signal processing tasks; (2) high-level programming abstraction; (3) heterogeneity and multi-platform interoperability; (4) easy and fast functionality extensibility; (5) quick porting procedure towards new C-like sensor platforms; (6) efficiency in resource-constrained embedded environments.

As second contribution, we have presented SPINE-*, a framework for rapid prototyping of WSN applications with autonomic characteristics. The SPINE-* architecture has been obtained as enhancement of SPINE2 by adding an autonomic plane including autonomic tasks. This implies that the original framework logic has been maintained unchanged, i.e. the autonomic plane can be designed by using the same task-oriented paradigm. An important advantage of such an architecture is that the autonomic capabilities can be easily added by guaranteeing the necessary separation of concerns between the business application logic and the autonomic plane. This characteristic facilitates the development as well as maintenance of complex applications, focusing on their high-level logic rather than having to cater for management and control functionality. This can be incorporated as an "add-on", during or even after the development of the main application. To validate our approach we have, in fact, added a number of autonomic tasks directly on top of the existing framework SPINE2. In this way we have been able to assess the ease of execution of our approach in a practical setting. In particular, a human

[1] *http://spine.deis.unical.it*

activity recognition application has been enhanced by inserting some autonomic elements providing the necessary operations for satisfying important self-* properties: data corruption detection, application reconfiguration, and context-aware privacy. Moreover, we have studied the impact of data faults on this BSN system, analyzed in which measure its recognition accuracy decreases under different kinds of data-fault models affecting the original raw-data accelerometer readings and then applied proper autonomic elements providing the necessary self-healing operations through which it has been possible to improve the accuracy of the original application.

As third contribution, the motivations and the benefits of using mobile agents in the WSN domain have been first introduced. Then, an innovative agent-oriented programming framework specifically designed for Java-based WSN, MAPS, has been presented. Moreover, a modified lightweight version of MAPS, TinyMAPS, for supporting the Sentilla JCreate sensor platform, has been also briefly described. MAPS allows developers to easily build WSN applications as a set of interacting stationary and mobile agents distributed on the sensor nodes (specifically Sun SPOTs). Such agents are supported by the MAPS agent execution engine which provides all the basic services including message transmission, agent creation, agent cloning, agent migration, timer handling, and easy access to the sensor node resources. In particular, the agent programming model adopted by MAPS is based on a multi-plane state machine (MPSM) whose state transition is controlled by Event-Condition-Action (ECA) rules, wheres the mobile agents interact through events that are asynchronously delivered by a dedicated component of MAPS. Such a MPSM-based agent behavior programming allows exploiting the benefits deriving from three main paradigms for WSN programming: event-driven programming, state-based programming and mobile agent-based programming. MAPS programming approach has been exemplified through a simple yet effective example conceived for carrying out some performance evaluations and a more specific case study involving agents in a human activity monitoring system. These examples clearly show how simple and rapid is the development of applications in MAPS and, in particular, the definition of agents' behavior by means of the state machine formalism and the Java-based MAPS API. And this demonstrates the effectiveness and suitability of MAPS to deal with the programming of complex applications, also with respect to other agent-based frameworks for WSNs, like AFME. The performance evaluation carried out on both the presented platform shows some performance penalties of MAPS with respect to AFME, and TinyMAPS with respect to MAPS. This is mainly due to the very time-consuming operations (Isolate hibernation/serialization and radiostream based communications) provided by the Sun SPOT SquawkVM, on which MAPS relies, and the very resource-constrained platform (Sentilla JCreate), on which TinyMAPS runs.

As fourth contribution, specific self-adapting protocols have been proposed. The CG-AODV routing protocol has been conceived as an enhancement

of AODV by introducing a novel "node concentration-driven gossiping" mechanism. A new measurable quantity for characterizing a WSN, called "node concentration" has been first introduced as a new measurable quantity for characterizing a WSN. It considers the density of a neighborhood by taking into account also the communication capabilities of nodes. Then, some simulations have been carried out for analyzing how the control traffic generated by a reactive routing protocol (i.e. AODV) impacts the network performance with respect to different values of node concentration. On the basis of the obtained results, a specific approach which introduces a mechanism to limit flooding, and then reduce the protocol overhead, through the use of a gossip-based filtering/propagation decision has been proposed and experimentally evaluated. Simulations results have shown that CG-AODV provides significant improvements in terms of packet delivery ratio and path discovery delay. The QL-MAC protocol, based on Q-Learning, has been designed such that each node independently determines an efficient wake-up schedule in order to limit as much as possible the number of slots in which the radio is turned on. Such a non-fixed and adaptive duty-cycle reduces the energy consumption over the time without affecting the other network performances. In particular, the simulation results show that, compared to other standard MAC protocols for WSNs, the adaptive behavior of QL-MAC guarantees better network performances with respect to both the packet delivery ratio and the average energy consumption. Moreover, the learning approach requires minimal overhead and very low computational complexity which are fundamental requirements in a resource-constrained embedded platforms like the ones constituting a WSN.

5.2 Future Directions

In the development of this thesis several issues emerged which deserve further examination in the future.

With regards to the SPINE2 framework, future work aims at improving it by broadening the current supported set of tasks with new ones, so as to increase the framework capabilities by providing new functions and algorithms to the developers so to better support the development of more complex data analysis applications and customized features. Similarly, SPINE-* will be extended to support developers in defining autonomic behaviors in their WSN applications by means of further tasks implementing specific self-* capabilities.

The current implementation of the task-oriented paradigm needs developers to specify on which specific WSN node each task has to be instantiated. Thus, an important improvement will be the introduction of particular high-level constructs able to manage a subset of nodes as a whole, so as to avoid the annoying work of assigning each task-node pair in large scale networks.

Moreover, since both framework are based on the same platform-independent middleware architecture, work is underway to port SPINE2/SPINE-* onto new sensor platforms so as to support a more heterogeneous WSN ecosystem.

As for the MAPS framework, further efforts will be devoted for optimizing its communication and migration mechanisms whose performance penalties emerged during the analysis. Moreover, further research will aim to explore the viability and the benefits of integrating the agent-oriented paradigm with the task-based approach provided by SPINE2/SPINE-*.

Future research will also focus on the definition of both high-level (e.g. application gateways) and low-level (IEEE 802.15.4 layer) solutions for agent communication interoperability between MAPS and AFME that would enable the development of heterogeneous agent-based WSN applications.

With respect to the specific self-adapting protocols proposed in this thesis, other simulations will be carried out for analyzing their behavior in further network scenarios and obtained results will be compared with other state-of-the-art protocols. In particular, it would be interesting to investigate to what extent node mobility and other different packet traffic patterns influence their performance.

Moreover, further reward functions will be defined and tested in the Q-Learning algorithm adopted by the QL-MAC protocol, by taking into consideration other network information, e.g. the nodes link quality.

Furthermore, by simulating WSNs adopting at the same time CG-AODV as routing protocol and QL-MAC as MAC layer, their reciprocal influence will be analyzed for figuring out how they can be modified so as to be able to directly collaborate. In such a way, it may be possible to test to what extent a cross-layering approach for self-adapting protocols is beneficial to global WSN networking performance.

Finally, an integrated approach among the contributions provided in this thesis will be investigated. Specifically, a generic autonomic WSN architecture is under consideration by integrating the high-level approaches for defining autonomic behavior in WSN applications and the low-level self-adapting protocols for supporting nodes' communication. The final aim is to provide a general-purpose framework for developing applications in generic WSNs supported by autonomic capabilities, where (i) the programming constructs allow developers to define both the application business logic and the high-level policies enabling the self-* properties according to the application's characteristics, whereas (ii) the low-level networking protocols are able to self-adapting their behavior on the basis of the application's communication needs.

5.3 Publications related with this Thesis

The research work related to this thesis has resulted in 1 co-authored book and 17 publications. Among them, there are 6 journal articles, 9 conference papers, and 2 book chapters.

5.3.1 Co-Authored book

– **Wearable Systems and Body Sensor Networks: from modeling to implementation**:
G. Fortino, S. Galzarano, R. Gravina. "Wearable Systems and Body Sensor Networks: from modeling to implementation", Wiley, USA, 2014, to appear.

5.3.2 Journal Articles

– **A task-oriented portable embedded framework for networked wearable computing**:
S. Galzarano and G. Fortino. A task-oriented portable embedded framework for networked wearable computing. Submitted to *ACM Transactions on Embedded Computing Systems (TECS)*. [Related to Chap. 3, Section 3.2 and 3.3]

– **A Framework for Collaborative Computing and Multi-Sensor Data Fusion in Body Sensor Networks**:
G. Fortino, S. Galzarano, R. Gravina, W. Li. A Framework for Collaborative Computing and Multi-Sensor Data Fusion in Body Sensor Networks. *Journal of Information Fusion*, Elsevier. Minor revision.

– **From Modeling to Implementation of Virtual Sensors in Body Sensor Networks** [131]:
N. Raveendranathan, S. Galzarano, V. Loseu, R. Gravina, R. Giannantonio, M. Sgroi, R. Jafari, and G. Fortino. From modeling to implementation of virtual sensors in body sensor networks. *IEEE Sensors Journal*, 12(3):583-593, March 2012. [Related to Chap. 3, Section 3.3.2]

– **A Task-based Architecture for Autonomic Body Sensor Networks** [132]:
S. Galzarano, G. Fortino, and A. Liotta. A task-based architecture for autonomic body sensor networks. *International Transactions on Systems Science and Applications*, 7(1/2):140-151, November 2011. [Related to Chap. 3, Section 3.4]

– **An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks** [96]:

F. Aiello, F.L. Bellifemine, G. Fortino, S. Galzarano, and R. Gravina. An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks. *Engineering Applications of Artificial Intelligence*, 24(7):1147-1161, October 2011. [Related to Chap. 3, Section 3.6]

– **An analysis of java-based mobile agent platforms for wireless sensor networks** [95]:

F. Aiello, G. Fortino, S. Galzarano, R. Gravina, and A. Guerrieri. An analysis of java-based mobile agent platforms for wireless sensor networks. *Multiagent and Grid Systems*, 7(6):243-267, January 2011. [Related to Chap. 3, Section 3.6]

5.3.3 Conference Papers

– **QL-MAC: a Q-Learning based MAC for Wireless Sensor Networks** [133]:

S. Galzarano, A. Liotta, and G. Fortino. QL-MAC: a Q-Learning based MAC for Wireless Sensor Networks. In Giancarlo Fortino, Rocco Aversa, Joanna Kolodziej, Jun Zhang, and Flora Amato, editors, *Advances of Parallel and Distributed Computing, Springer, LNCS series, Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2013)*, Lecture Notes on Computer Science (LNCS), Springer, Vietri sul Mare, Italy, 2013. Springer Berlin Heidelberg. [Related to Chap. 4, Section 4.2]

– **Gossiping-based AODV for Wireless Sensor Networks** [134]:

S. Galzarano, C. Savaglio, A. Liotta, and G. Fortino. Gossiping-based AODV for Wireless Sensor Networks. In *Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Manchester, UK, 2013. [Related to Chap. 4, Section 4.1]

– **An autonomic plane for Wireless Body Sensor Networks** [135]:

G. Fortino, S. Galzarano, and A. Liotta. An autonomic plane for Wireless Body Sensor Networks. In *International Conference on Computing, Networking and Communications (ICNC), 2012*, pages 94-98, Maui, Hawaii, February 2012. [Related to Chap. 3, Section 3.4]

– **Embedded self-healing layer for detecting and recovering sensor faults in body sensor networks** [136]:

S. Galzarano, G. Fortino, and A. Liotta. Embedded self-healing layer for de- tecting and recovering sensor faults in body sensor networks. In *Proceedings of the 2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2377-2382, 2012. [Related to Chap. 3, Section 3.5]

– **Integrating Jade and MAPS for the development of Agent-based WSN applications** [137]:

M. Mesjasz, D. Cimadoro, S. Galzarano, M. Ganzha, G. Fortino, and M. Paprzycki. Integrating jade and MAPS for the development of agent-based WSN applications. In *Intelligent Distributed Computing VI, Springer, SCI se- ries, Proceedings of the 6th International Symposium on Intelligent Distributed Computing (IDC 2012)*, volume 446 of *Studies in Computational Intelligence*, pages 211-220, Rende, Italy, 2013. Springer Berlin / Heidelberg. [Related to Chap. 3, Section 3.6]

– **Human Postures Recognition Based on D-S Evidence Theory and Multi-sensor Data Fusion** [138]:

W. Li, J. Bao, X. Fu, G. Fortino, and S. Galzarano. Human Postures Recognition Based on D-S Evidence Theory and Multi-sensor Data Fusion. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 912-917, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

– **TinyMAPS: A Lightweight Java-Based Mobile Agent System for Wireless Sensor Networks** [139]:

F. Aiello, G. Fortino, S. Galzarano, and A. Vittorioso. TinyMAPS: a light- weight Java-Based Mobile Agent System for Wireless Sensor Networks. In F. Brazier, Kees Nieuwenhuis, Gregor Pavlin, Martijn Warnier, and Costin Badica, editors, *Intelligent Distributed Computing V, Springer, SCI series, Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*, volume 382 of *Studies in Computational Intelligence*, pages 161-170, Delft, The Netherlands, 2012. Springer Berlin / Heidelberg. [Related to Chap. 3, Section 3.6]

– **Collaborative Body Sensor Networks** [140]:

A. Augimeri, G. Fortino, S. Galzarano, and R. Gravina. Collaborative body sensor networks. In *Proceedings of the 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3427-3432, Anchorage, Alaska, October 2011.

– **Agent-based Development of Wireless Sensor Network Applications** [141]:

G. Fortino, S. Galzarano, R. Gravina, and A. Guerrieri. Agent-based Development of Wireless Sensor Network Applications. In *Proceedings of 12th Workshop on Objects and Agents (WOA 2011)*, Rende (CS), July 2011. [Related to Chap. 3, Section 3.6]

5.3.4 Book Chapters

– **On the development of mobile agent systems for wireless sensor networks: issues and solutions** [142]:

G. Fortino and S. Galzarano. On the development of mobile agent systems for wireless sensor networks: issues and solutions. In Maria Ganzha and Lakhmi Jain, editors, *Multiagent*

Systems and Applications: Practice and Experience, number 45 in Intelligent Systems Reference Library, pages 185-215. Springer- Verlag Berlin Heidelberg, 2013. [Related to Chap. 3, Section 3.6]

– **Signal processing in-node frameworks for Wireless Body Sensor Networks: from low-level to high-level approaches** [143]:
F. Aiello, G. Fortino, S. Galzarano, R. Gravina, and A. Guerrieri. Signal processing in-node frameworks for Wireless Body Sensor Networks: from low- level to high-level approaches. In Mehmet R. Yuce and Jamil Y. Khan, editors, *Wireless Body Area Networks: Technology, Implementation and Applications*, pages 107-135. Pan Stanford Publishing, 2011. [Related to Chap. 3, Section 3.6]

Acknowledgments

I would like to thank all those who have helped and supported me during my PhD, my supervisors, my family, my love, my colleagues and my friends.

La presente tesi è cofinanziata con il sostegno della Commissione Europea, Fondo Sociale Europeo e della Regione Calabria.
L'autore è il solo responsabile di questa tesi e la Commissione Europea e la Regione Calabria declinano ogni responsabilitá sull'uso che potrá essere fatto delle informazioni in essa contenute.

References

1. Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330, 2008.
2. Kazem Sohraby, Daniel Minoli, and Taieb Znati. *Wireless Sensor Networks: Technology, Protocols, and Applications*. Wiley-Interscience, 1 edition, April 2007.
3. Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.
4. David Marsh, Richard Tynan, Donal O'Kane, and Gregory M. P. O'Hare. Autonomic wireless sensor networks. *Engineering Applications of Artificial Intelligence*, 17(7):741–748, 2004. Autonomic Computing Systems.
5. Min Chen, Sergio Gonzalez, Athanasios Vasilakos, Huasong Cao, and Victor C. M. Leung. Body area networks: A survey. *Mobile Networks and Applications*, 16(2):171–193, April 2011.
6. Guang-Zhong Yang. *Body Sensor Networks*. Springer, 2006.
7. A. Pantelopoulos and N.G. Bourbakis. A survey on wearable sensor-based systems for health monitoring and prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(1):1–12, January 2010.
8. T. Shany, S.J. Redmond, M.R. Narayanan, and N.H. Lovell. Sensors-based wearable systems for monitoring of human movement and falls. *IEEE Sensors Journal*, 12(3):658–670, March 2012.
9. Yang Hao and Robert Foster. Wireless body sensor networks for health-monitoring applications. *Physiological measurement*, 29(11):27–56, November 2008. PMID: 18843167.
10. A. Augimeri, G. Fortino, M.R. Rege, V. Handziski, and A. Wolisz. A cooperative approach for handshake detection based on body sensor networks. In *2010 IEEE International Conference on Systems Man and Cybernetics (SMC)*, pages 281–288, October 2010.
11. Sergio Guilln, Maria Teresa Arredondo, and Elena Castellano. A survey of commercial wearable systems for sport application. In Annalisa Bonfiglio and Danilo De Rossi, editors, *Wearable Monitoring Systems*, pages 165–178. Springer US, January 2011.

12. N. Gupta and S. Jilla. Digital fitness connector: Smart wearable system. In *2011 First International Conference on Informatics and Computational Intelligence (ICI)*, pages 118–121, December 2011.
13. Hagop Hagopian Bobak Mortazavi. A wireless body-wearable sensor system for designing physically interactive video games. In *Proceedings of the International Conference on Biomedical Electronics and Devices*, pages 62–69, Rome, Italy, 2011.
14. Tsutomu Terada and Kohei Tanaka. A framework for constructing entertainment contents using flash and wearable sensors. In *Proceedings of the 9th international conference on Entertainment computing*, ICEC'10, pages 334–341, Berlin, Heidelberg, 2010. Springer-Verlag.
15. Jiung-Yao Huang and Chung-Hsien Tsai. A wearable computing environment for the security of a large-scale factory. In *Proceedings of the 12th international conference on Human-computer interaction: interaction platforms and techniques*, HCI'07, pages 1113–1122, Berlin, Heidelberg, 2007. Springer-Verlag.
16. Hakima Chaouchi, editor. *The Internet of Things: Connecting Objects*. Wiley-ISTE, Hoboken, NJ, London, 2010.
17. Mari Domingo. A context-aware service architecture for the integration of body sensor networks and social networks through the IP multimedia subsystem. *IEEE Communications Magazine*, 49(1):102–108, January 2011.
18. L. Mainetti, L. Patrono, and A. Vilei. Evolution of wireless sensor networks towards the internet of things: A survey. In *2011 19th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, 2011.
19. Roy Sterritt and D. Bustard. Autonomic computing - a means of achieving dependability? In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 247–251, 2003.
20. Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM T.J. Watson Labs, NY, 2001.
21. IBM Corp. An architectural blueprint for autonomic computing. Technical report, IBM Corp., USA, October 2004.
22. B. Melcher and B. Mitchell. Towards an autonomic framework: Self-configuring network services and developing autonomic applications. *Intel Technical Journal*, 8(1):279–290, November 2004.
23. J.P. Bigus, D.A. Schlosnagle, J. R. Pilgrim, W.N. Mills III, and Y. Diao. ABLE: a toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.
24. G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 22–30, 2003.
25. Giuseppe Valetto and Gail Kaiser. Using process technology to control and coordinate software adaptation. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 262–272, Washington, DC, USA, 2003. IEEE Computer Society.
26. F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. 2K: a distributed operating system for dynamic heterogeneous environments. In *The Ninth International Symposium on High-Performance Distributed Computing, 2000. Proceedings*, pages 201–208, 2000.

27. Dharini Balasubramaniam, Ron Morrison, Graham Kirby, Kath Mickan, Brian Warboys, Ian Robertson, Bob Snowdon, R. Mark Greenwood, and Wykeen Seet. A software architecture approach for structuring autonomic systems. In *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, DEAS '05, pages 1–7, New York, NY, USA, 2005. ACM.
28. Thomas Hamann, Gerald Hbsch, and Thomas Springer. A model-driven approach for developing adaptive software systems. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'08, pages 196–209, Berlin, Heidelberg, 2008. Springer-Verlag.
29. R. Cervenka, D. Greenwood, and I. Trencansky. The AML approach to modeling autonomic systems. In *2006 International Conference on Autonomic and Autonomous Systems, 2006. ICAS '06*, page 29, July 2006.
30. Yuan-Shun Dai, Tom Marshall, and Xiaohong Guan. Autonomic and dependable computing: Moving towards a model-driven approach. *Journal of Computer Science*, 2(6):496–504, June 2006.
31. Nancy Samaan and Ahmed Karmouch. Towards autonomic network management: an analysis of current and future research directions. *IEEE Communications Surveys Tutorials*, 11(3):22–36, 2009.
32. L. B. Ruiz, J. M. Nogueira, and A. A. F. Loureiro. MANNA: a management architecture for wireless sensor networks. *Communications Magazine, IEEE*, 41(2):116–125, 2003.
33. Hyungjoo Song, Daeyoung Kim, Kangwoo Lee, and Jongwoo Sung. UPnP-based sensor network management architecture. In *Second International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2005)*, 2005.
34. Winnie L. Lee, Amitava Datta, and Rachel Cardell-Oliver. WinMS: Wireless sensor Network-Management system, an adaptive Policy-Based management for wireless sensor networks. Technical report, The University of Western Australia, June 2006.
35. Themistoklis Bourdenas and Morris Sloman. Starfish: policy driven self-management in wireless sensor networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 75–83, New York, NY, USA, 2010. ACM.
36. Lilia Paradis and Qi Han. A survey of fault management in wireless sensor networks. *J. Netw. Syst. Manage.*, 15(2):171–190, June 2007.
37. Pruet Boonma and Junichi Suzuki. Bisnet: A biologically-inspired middleware architecture for self-managing wireless sensor networks. *Computer Networks*, 51(16):4599 – 4616, 2007.
38. Myeong-Hyeon Lee and Yoon-Hwa Choi. Fault detection of wireless sensor networks. *Comput. Commun.*, 31(14):3469–3475, September 2008.
39. V. Turau and C. Weyer. Fault tolerance in wireless sensor networks through self-stabilisation. *Int. J. Commun. Netw. Distrib. Syst.*, 2(1):78–98, November 2009.
40. Hayoung Oh, Inshil Doh, and Kijoon Chae. A fault management and monitoring mechanism for secure medical sensor network. *IJCSA*, pages 43–56, 2009.
41. Muhammad Asim, Hala Mokhtar, and Madjid Merabti. A self-managing fault management mechanism for wireless sensor networks. *International Journal of Wireless Mobile Networks*, 2(4):14, 2010.
42. Peng Jiang. A new method for node fault detection in wireless sensor networks. *Sensors*, 9(2):1282–1294, 2009.

43. Jae-Young Choi, Sung-Jib Yim, Yoon Jae Huh, and Yoon-Hwa Choi. An adaptive fault detection scheme for wireless sensor networks. In *Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems*, pages 106–110, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).
44. Mengjie Yu, Hala Mokhtar, and Madjid Merabti. Self-managed fault management in wireless sensor networks. In *Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, UBICOMM '08, pages 13–18, Washington, DC, USA, 2008. IEEE Computer Society.
45. Sai Ji, Shen-fang Yuan, Ting-huai Ma, and Chang Tan. Distributed fault detection for wireless sensor based on weighted average. In *Proceedings of the 2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, NSWCTC '10, pages 57–60, Washington, DC, USA, 2010. IEEE Computer Society.
46. Min Ding, Dechang Chen, Kai Xing, and Xiuzhen Cheng. Localized fault-tolerant event boundary detection in sensor networks. In *In Proc. of IEEE INFOCOM*, pages 902–913, 2005.
47. Themistoklis Bourdenas and Morris Sloman. Towards self-healing in wireless sensor networks. In *Proceedings of the 2009 Sixth International Workshop on Wearable and Implantable Body Sensor Networks*, BSN '09, pages 15–20, Washington, DC, USA, 2009. IEEE Computer Society.
48. R.V. Kulkarni, A. Forster, and G.K. Venayagamoorthy. Computational intelligence in wireless sensor networks: A survey. *Communications Surveys Tutorials, IEEE*, 13(1):68 –96, 2011.
49. S.S. Iyengar, N. Parameshwaran, V.V. Phoha, N. Balakrishnan, and C.D. Okoye. *Fundamentals of Sensor Network Programming: Applications and Technology*. Wiley, 2010.
50. Bartolome Rubio, Manuel Diaz, and Jose M. Troya. Programming approaches and challenges for wireless sensor networks. In *Proceedings of the Second International Conference on Systems and Networks Communications*, ICSNC '07, pages 36–, Washington, DC, USA, 2007. IEEE Computer Society.
51. G. Fortino, R. Giannantonio, R. Gravina, P. Kuryloski, and R. Jafari. Enabling effective programming and flexible management of efficient body sensor network applications. *Human-Machine Systems, IEEE Transactions on*, 43(1):115–133, 2013.
52. Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: an operating system for sensor networks. In *Ambient Intelligence*. Springer Verlag, 2004.
53. Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.
54. Mohammad Mostafizur Rahman Mozumdar, Luciano Lavagno, and Laura Vanzago. A comparison of software platforms for wireless sensor networks: Mantis, tinyos, and zigbee. *ACM Trans. Embed. Comput. Syst.*, 8(2):12:1–12:23, February 2009.

55. G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010.
56. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
57. Johannes Gehrke and Praveen Seshadri. Querying the physical world. *IEEE Personal Communications*, 7:10–15, 2000.
58. Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. In *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.
59. F. Aiello, G. Fortino, R. Gravina, and A. Guerrieri. A java-based agent platform for programming wireless sensor networks. *The Computer Journal*, 54(3):439–454, 2011.
60. C. Muldoon, G. M. P. O'hare, R. Collier, and M. J. O'grady. Agent factory micro edition: A framework for ambient applications. In *Proceedings of Intelligent Agents in Computing Systems Workshop (held in Conjunction with International Conference on Computational Science (ICCS)) Reading, UK. Lecture Notes in Computer Science (LNCS*, pages 727–734. Springer-Verlag Publishers, 2006.
61. Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Trans. Auton. Adapt. Syst.*, 4(3):1–26, 2009.
62. Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 187–200, New York, NY, USA, 2003. ACM.
63. YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1297–1300, New York, NY, USA, 2006. ACM.
64. Fabio Bellifemine, Giancarlo Fortino, Roberta Giannantonio, Raffaele Gravina, Antonio Guerrieri, and Marco Sgroi. Spine: a domain-specific framework for rapid prototyping of wbsn applications. *Softw. Pract. Exper.*, 41(3):237–265, March 2011.
65. Clemens Lombriser, Daniel Roggen, Mathias Stäger, and Gerhard Tröster. Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. In Torsten Braun, Georg Carle, and Burkhard Stiller, editors, *15. Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pages 127–138. Springer Berlin Heidelberg, 0 2007.
66. Mi Zhang and A.A. Sawchuk. A customizable framework of body area sensor network for rehabilitation. In *Applied Sciences in Biomedical and Communication Technologies, 2009. ISABEL 2009. 2nd International Symposium on*, pages 1–6, nov. 2009.
67. G. Fortino, A. Guerrieri, G.M.P. O'Hare, and A. Ruzzelli. A flexible building management framework based on wireless sensor and actuator networks. *Journal of Network and Computer Applications*, 35(6):1934–1952, November 2012.

68. K. Lorincz, D.J. Malan, T.R.F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: challenges and opportunities. *Pervasive Computing, IEEE*, 3(4):16–23, oct. 2004.
69. Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, Berkeley, CA, USA, 2005. USENIX Association.
70. Luca Mottola and Gian Pietro Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In Phillip B. Gibbons, Tarek Abdelzaher, James Aspnes, and Ramesh Rao, editors, *Distributed Computing in Sensor Systems*, number 4026 in Lecture Notes in Computer Science, pages 150–168. Springer Berlin Heidelberg, January 2006.
71. Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, New York, NY, USA, 2005. ACM.
72. R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498, april 2007.
73. M.M.R. Mozumdar, L. Lavagno, L. Vanzago, and A.L. Sangiovanni-Vincentelli. Hilac: A framework for hardware in the loop simulation and multi-platform automatic code generation of wsn applications. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 88 –97, july 2010.
74. W.B. Heinzelman, A.L. Murphy, H.S. Carvalho, and M.A. Perillo. Middleware to support sensor network applications. *Network, IEEE*, 18(1):6–14, jan/feb 2004.
75. Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *SIGARCH Comput. Archit. News*, 30(5):85–95, 2002.
76. Sam Michiels, Wouter Horr, Wouter Joosen, and Pierre Verbaeten. DAViM: a dynamically adaptable virtual machine for sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 7–12, New York, NY, USA, 2006. ACM.
77. Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *Proceedings of the 6th ACM and IEEE International conference on Embedded software*, EMSOFT '06, pages 112–121, New York, NY, USA, 2006. ACM.
78. Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.
79. P.J. Marron, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. Tinycubus: a flexible and adaptive framework sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 278–289, 2005.
80. Viktor K. Prasanna and Amol B. Bakshi. *Architecture-independent Programming for Wireless Sensor Networks: An Architecture-independent Approach (Wiley Series on Parallel and Distributed Computing)*. John Wiley & Sons, 2008.

81. Pasquale Panuccio, Hassan Ghasemzadeh, Giancarlo Fortino, and Roozbeh Jafari. Power-aware action recognition with optimal sensor selection: an AdaBoost driven distributed template matching approach. In *Proceedings of the First ACM Workshop on Mobile Systems, Applications, and Services for Healthcare*, mHealthSys '11, pages 51–56, New York, NY, USA, 2011. ACM.
82. Eric Guenterberg, Hassan Ghasemezadeh, and Roozbeh Jafari. A distributed hidden markov model for fine-grained annotation in body sensor networks. In *The Sixth International Workshop on Body Sensor Networks (BSN)*, 2009.
83. G. Fortino, A. Guerrieri, F. Bellifemine, and R. Giannantonio. SPINE2: developing BSN applications on heterogeneous sensor nodes. In *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, pages 128–131, jul. 2009.
84. R. Gravina, A. Guerrieri, G. Fortino, F. Bellifemine, R. Giannantonio, and M. Sgroi. Development of body sensor network applications using SPINE. In *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pages 2810–2815, 2008.
85. Alex Rogers, Daniel D. Corkill, and Nicholas R. Jennings. Agent technologies for sensor networks. *IEEE Intelligent Systems*, 24:13–17, 2009.
86. Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, March 1999.
87. Meritxell Vinyals, Juan A. Rodríguez-Aguilar, and Jesús Cerquides. A survey on sensor networks from a multi-agent perspective. *The Computer Journal*, 54(3):455–470, 2010.
88. Syed Rehan Afzal, Christophe Huygens, and Wouter Joosen. Extending middleware frameworks for wireless sensor networks. In Syed Rehan Afzal, editor, *Ultra Modern Telecommunications & Workshops, 2009 (ICUMT)*, pages 1–7. IEEE, December 2009.
89. Francesco Aiello, Giancarlo Fortino, and Antonio Guerrieri. Using mobile agents as enabling technology for wireless sensor networks. *Sensor Technologies and Applications, International Conference on*, pages 549–554, 2008.
90. Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Trans. Auton. Adapt. Syst.*, 4(3):1–26, July 2009.
91. S. Suenaga and S. Honiden. Enabling direct communication between mobile agents in wireless sensor networks. In *1st Int'l Workshop on Agent Technology for Sensor Networks (ATSN-07), jointly held with 6th Int'l Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-07), Honolulu, Hawaii, 14th May*, 2007.
92. L. Szumel, J. LeBrun, and J. D. Owens. Towards a mobile agent framework for sensor networks. In *Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 79–87, Washington, DC, USA, 2005. IEEE Computer Society.
93. Ramon Lopes, Flavio Assis, and Carlos Montez. MASPOT: A Mobile Agent System for Sun SPOT. In *Proceedings of the 2011 Tenth International Symposium on Autonomous Decentralized Systems*, ISADS '11, pages 25–31, Washington, DC, USA, 2011. IEEE Computer Society.
94. Marios D. Dikaiakos, Melinos Kyriakou, and George Samaras. Performance evaluation of mobile-agent middleware: A hierarchical approach. In *Proceedings of the 5th International Conference on Mobile Agents*, MA '01, pages 244–259, London, UK, 2002. Springer-Verlag.

95. F. Aiello, G. Fortino, S. Galzarano, R. Gravina, and A. Guerrieri. An analysis of java-based mobile agent platforms for wireless sensor networks. *Multiagent and Grid Systems*, 7(6):243–267, January 2011.
96. F. Aiello, F.L. Bellifemine, G. Fortino, S. Galzarano, and R. Gravina. An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks. *Engineering Applications of Artificial Intelligence*, 24(7):1147–1161, October 2011.
97. T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.
98. Uwe Maurer, Asim Smailagic, Daniel P. Siewiorek, and Michael Deisher. Activity recognition and monitoring using multiple sensors on different body positions. In *BSN '06: Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, pages 113–116, Washington, DC, USA, 2006. IEEE Computer Society.
99. Nikolaos A. Pantazis, Stefanos A. Nikolidakis, and Dimitrios D. Vergados. Energy-efficient routing protocols in wireless sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 15(2):551–591, 2013.
100. U. Prathap, D.P. Shenoy, K.R. Venugopal, and L.M. Patnaik. Wireless sensor networks applications and routing protocols: Survey and research challenges. In *2012 Int'l Symp. on Cloud and Services Computing (ISCOS)*, pages 49–56, 2012.
101. Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349, May 2005.
102. J.N. Al-Karaki and A.E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11(6):6–28, 2004.
103. C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *Second IEEE Workshop on Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99*, pages 90–100, 1999.
104. C.E. Perkins, E.M. Royer, and S. Das. Ad-hoc on-demand distance vector routing. *RFC 3561*, July 2003.
105. Demin Wang, Yi Cheng, Yun Wang, and D.P. Agrawal. Lifetime enhancement of wireless sensor networks by differentiable node density deployment. In *IEEE Conf. on Mobile Adhoc and Sensor Systems*, pages 546–549, 2006.
106. Mu-Huan Chiang and G.T. Byrd. Neighborhood-aware density control in wireless sensor networks. In *IEEE Int'l Conf. on Sensor Networks, Ubiquitous and Trustworthy Computing (SUTC 08)*, pages 122–129, 2008.
107. Moez Esseghir and Nizar Bouabdallah. Node density control for maximizing wireless sensor network lifetime. *International Journal of Network Management*, 18(2):159–170, 2008.
108. Bang Wang, Kee Chaing Chua, Vikram Srinivasan, and Wei Wang. Sensor density for complete information coverage in wireless sensor networks. In Kay Romer, Holger Karl, and Friedemann Mattern, editors, *Wireless Sensor Networks*, number 3868 in Lecture Notes in Computer Science, pages 69–82. Springer, January 2006.
109. J.C. Melo and L.B. Ruiz. Data-centric density control for wireless sensor networks. In *The Fourth Int'l Conf. on Wireless and Mobile Communications, 2008. ICWMC '08*, pages 19–24, 2008.
110. Ruay-Shiung Chang and Shuo-Hung Wang. Self-deployment by density control in sensor networks. *IEEE Transactions on Vehicular Technology*, 57(3):1745–1755, 2008.

111. Yang Shen, Xianglan Yin, Hua Chen, Wangdong Qi, and Hao Dai. A density control algorithm for surveillance sensor networks. In Xiuzhen Cheng, Wei Li, and Taieb Znati, editors, *Wireless Algorithms, Systems, and Applications*, number 4138 in Lecture Notes in Computer Science, pages 199–205. Springer Berlin Heidelberg, January 2006.
112. J. Rahman, M.A.M. Hasan, and M.K.B. Islam. Comparative analysis the performance of AODV, DSDV and DSR routing protocols in wireless sensor network. In *2012 7th Int'l Conf. on Electrical Computer Engineering (ICECE)*, pages 283–286, 2012.
113. M. Kassim, R.A. Rahman, and R. Mustapha. Mobile ad hoc network routing protocols comparison for wireless sensor network. In *2011 IEEE Int'l Conf. on System Engineering and Technology*, pages 148–152, 2011.
114. M. Pandey and S. Verma. Performance evaluation of AODV for different mobility conditions in WSN. In *Int'l Conf. on Multimedia, Signal Processing and Comm. Technologies (IMPACT)*, pages 240–243, 2011.
115. M.N. Jambli, K. Zen, H. Lenando, and A. Tully. Performance evaluation of AODV routing protocol for mobile wireless sensor network. In *7th Int'l Conf. on Information Technology in Asia (CITA 11)*, pages 1–6, 2011.
116. Canfeng Chen and Jian Ma. Simulation study of AODV performance over IEEE 802.15.4 MAC in WSN with mobile sinks. In *21st Int'l Conf. on Advanced Information Networking and Applications Workshops, 2007, AINAW '07*, volume 2, pages 159–164, 2007.
117. Shuangyin Ren, Huanmin Han, Baoliang Li, Jia Lu, Cgao Peng, and Wenhua Dou. An improved wireless sensor networks routing protocol based on AODV. In *2012 IEEE 12th Int'l Conf. on Computer and Information Technology (CIT)*, pages 742–746, 2012.
118. Debajyoti Mishra, Ashima Rout, and Srinivas Sethi. An effective and scalable AODV for wireless ad hoc sensor networks. *International Journal of Computer Applications*, 5(4):33–38, August 2010.
119. Wei Li, Ming Chen, and Ming-ming Li. An enhanced AODV route protocol applying in the wireless sensor networks. In Bingyuan Cao, Tai-Fu Li, and Cheng-Yi Zhang, editors, *Fuzzy Information and Engineering Volume 2*, number 62 in Advances in Intelligent and Soft Computing, pages 1591–1600. Springer Berlin Heidelberg, January 2009.
120. Antonio Liotta and George Exarchakos. Reactive networks. In *Networks for Pervasive Services*, number 92 in LNEE, pages 65–78. Springer, January 2011.
121. Li Layuan, Li Chunlin, and Yaun Peiyan. Performance evaluation and simulations of routing protocols in ad hoc networks. *Computer Communications*, 30(8):1890–1898, June 2007.
122. Himabindu Pucha, Saumitra M. Das, and Y. Charlie Hu. The performance impact of traffic patterns on routing protocols in mobile ad hoc networks. *Computer Networks*, 51(12):3595–3616, August 2007.
123. Paul J.M. Havinga and Gerard J.M. Smit. Energy-efficient tdma medium access control protocol scheduling. In *Asian International Mobile Computing Conf., AMOC*, pages 1–10, 2000.
124. Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proc. 21st International Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, USA, 2002.

125. Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys 03)*, 2003.
126. Tao Zheng, S. Radhakrishnan, and V. Sarangan. Pmac: an adaptive energy-efficient mac protocol for wireless sensor networks. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005.
127. Zhenzhen Liu and Itamar Elhanany. RL-MAC: a reinforcement learning based MAC protocol for wireless sensor networks. *Int. J. Sen. Netw.*, 1(3/4):117–124, January 2006.
128. Mihail Mihaylov, Karl Tuyls, and Ann Now. Decentralized learning in wireless sensor networks. In *Proceedings of the Second international conference on Adaptive and Learning Agents*, ALA'09, pages 60–73, Berlin, Heidelberg, 2010. Springer-Verlag.
129. Yi Chu, P.D. Mitchell, and D. Grace. ALOHA and q-learning based medium access control for wireless sensor networks. In *2012 International Symposium on Wireless Communication Systems (ISWCS)*, pages 511–515, 2012.
130. Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
131. N. Raveendranathan, S. Galzarano, V. Loseu, R. Gravina, R. Giannantonio, M. Sgroi, R. Jafari, and G. Fortino. From modeling to implementation of virtual sensors in body sensor networks. *IEEE Sensors Journal*, 12(3):583 –593, March 2012.
132. S. Galzarano, G. Fortino, and A. Liotta. A task-based architecture for autonomic body sensor networks. *International Transactions on Systems Science and Applications*, 7(1/2):140–151, November 2011.
133. S. Galzarano, A. Liotta, and G. Fortino. QL-MAC: a Q-Learning based MAC for Wireless Sensor Networks. In Giancarlo Fortino, Rocco Aversa, Joanna Kolodziej, Jun Zhang, and Flora Amato, editors, *Advances of Parallel and Distributed Computing, Springer, LNCS series, Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2013)*, Lecture Notes on Computer Science (LNCS), Springer, Vietri sul Mare, Italy, 2013. Springer Berlin Heidelberg.
134. S. Galzarano, C. Savaglio, A. Liotta, and G. Fortino. Gossiping-based AODV for Wireless Sensor Networks. In *Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Manchester, UK, 2013.
135. G. Fortino, S. Galzarano, and A. Liotta. An autonomic plane for Wireless Body Sensor Networks. In *International Conference on Computing, Networking and Communications (ICNC), 2012*, pages 94–98, Maui, Hawaii, February 2012.
136. S. Galzarano, G. Fortino, and A. Liotta. Embedded self-healing layer for detecting and recovering sensor faults in body sensor networks. In *Proceedings of the 2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2377–2382, 2012.
137. M. Mesjasz, D. Cimadoro, S. Galzarano, M. Ganzha, G. Fortino, and M. Paprzycki. Integrating jade and MAPS for the development of agent-based WSN applications. In *Intelligent Distributed Computing VI, Springer, SCI series, Proceedings of the 6th International Symposium on Intelligent Distributed Computing (IDC 2012)*, volume 446 of *Studies in Computational Intelligence*, pages 211–220, Rende, Italy, 2013. Springer Berlin / Heidelberg.

138. W. Li, J. Bao, X. Fu, G. Fortino, and S. Galzarano. Human Postures Recognition Based on D-S Evidence Theory and Multi-sensor Data Fusion. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 912–917, Los Alamitos, CA, USA, 2012. IEEE Computer Society.
139. F. Aiello, G. Fortino, S. Galzarano, and A. Vittorioso. TinyMAPS: a lightweight Java-Based Mobile Agent System for Wireless Sensor Networks. In F. Brazier, Kees Nieuwenhuis, Gregor Pavlin, Martijn Warnier, and Costin Badica, editors, *Intelligent Distributed Computing V, Springer, SCI series, Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*, volume 382 of *Studies in Computational Intelligence*, pages 161–170, Delft, The Netherlands, 2012. Springer Berlin / Heidelberg.
140. A. Augimeri, G. Fortino, S. Galzarano, and R. Gravina. Collaborative body sensor networks. In *Proceedings of the 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3427–3432, Anchorage, Alaska, October 2011.
141. G. Fortino, S. Galzarano, R. Gravina, and A. Guerrieri. Agent-based Development of Wireless Sensor Network Applications. In *Proceedings of 12th Workshop on Objects and Agents (WOA 2011)*, Rende (CS), July 2011.
142. G. Fortino and S. Galzarano. On the development of mobile agent systems for wireless sensor networks: issues and solutions. In Maria Ganzha and Lakhmi Jain, editors, *Multiagent Systems and Applications: Practice and Experience*, number 45 in Intelligent Systems Reference Library, pages 185–215. Springer-Verlag Berlin Heidelberg, 2013.
143. F. Aiello, G. Fortino, S. Galzarano, R. Gravina, and A. Guerrieri. Signal processing in-node frameworks for Wireless Body Sensor Networks: from low-level to high-level approaches. In Mehmet R. Yuce and Jamil Y. Khan, editors, *Wireless Body Area Networks: Technology, Implementation and Applications*, pages 107–135. Pan Stanford Publishing, 2011.