SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:237–265
Published online 7 September 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.998

SPINE: A domain-specific framework for rapid prototyping of WBSN applications‡

Fabio Bellifemine[1], Giancarlo Fortino[2,*,†], Roberta Giannantonio[1], Raffaele Gravina[2,3], Antonio Guerrieri[2] and Marco Sgroi[3]

[1]*Telecom Italia, Turin, Italy*
[2]*Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), Italy*
[3]*WSN Lab Telecom Italia, Berkeley, CA, U.S.A.*

SUMMARY

Wireless body sensor networks (WBSNs) enable a broad range of applications for continuous and real-time health monitoring and medical assistance. Programming WBSN applications is a complex task especially due to the limitation of resources of typical hardware platforms and to the lack of suitable software abstractions. In this paper, SPINE (signal processing in-node environment), a domain-specific framework for rapid prototyping of WBSN applications, which is lightweight and flexible enough to be easily customized to fit particular application-specific needs, is presented. The architecture of SPINE has two main components: one implemented on the node coordinating the WBSN and one on the nodes with sensors. The former is based on a Java application, which allows to configure and manage the network and implements the classification functions that are too heavy to be implemented on the sensor nodes. The latter supports sensing, computing and data transmission operations through a set of libraries, protocols and utility functions that are currently implemented for TinyOS platforms. SPINE allows evaluating different architectural choices and deciding how to distribute signal processing and classification functions over the nodes of the network. Finally, this paper describes an activity monitoring application and presents the benefits of using the SPINE framework. Copyright © 2010 John Wiley & Sons, Ltd.

Received 8 June 2009; Revised 14 February 2010; Accepted 15 June 2010

KEY WORDS: body sensor networks; domain-specific frameworks; signal processing; SPINE

1. INTRODUCTION

A Wireless Sensor Network (WSN) [1] is a collection of tiny devices capable of sensing, computation and wireless communication operating in a certain environment to monitor and control events of interest in a distributed manner and collectively react to critical situations. WSN applications span various domains such as environmental and building monitoring and surveillance, pollution monitoring, agriculture, health care, home-automation, energy management, earthquake and eruption monitoring.

When applied to the monitoring of human body parameters, WSNs are usually called Wireless Body Sensor Networks (WBSNs) [2] or just BSNs and can significantly improve the quality of life by enabling continuous and real-time medical assistance at low cost. Health-care applications

*Correspondence to: Giancarlo Fortino, Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), Italy.
†E-mail: g.fortino@unical.it

‡This paper is a significantly extended version of R. Gravina, A. Guerrieri, G. Fortino, F. Bellifemine, R. Giannantonio, M. Sgrio, 'Development of Body Sensor Network Applications using SPINE,' In *Proc. of. IEEE International Conference on Systems, Man, and Cybernetics* (SMC 2008), Singapore, Oct. 12–15, 2008.

based on BSNs include early detection or prevention of diseases, elderly person assistance at home, e-fitness, rehabilitation after surgeries, motion and gestures detection, cognitive and emotional recognition, medical assistance in disaster events, etc.

Programming BSN applications is a complex task mainly due to the difficult resource constraints of wearable devices and to the lack of proper and easy to use software abstractions. Three main approaches have been developed for the design of BSN applications. The most common approach consists of developing prototype applications on BSN nodes as a monolithic block assembling low-level services, reusable components and application-specific logic. As a result, the software is poorly reusable and difficult to extend. Moreover, the risk of errors is significantly high and the debugging can be a very time-consuming process. The second approach is based on general-purpose middleware. Middleware is a software layer consisting of a set of services implemented across a network. It hides the complexities of low system and network layers and provides proper abstractions and interfaces to the upper layers. In this way application developers can focus on the application logic without dealing with the implementation details of the underlying services. As a consequence, the development time is generally shortened. Furthermore, if the middleware is well optimized, the overall system could even reach a higher performance. While in traditional distributed systems a number of general-purpose middlewares such as CORBA, DCOM and RMI have been widely used thanks to their ability to work well for very different applications, the current general-purpose middlewares for WSN (e.g. Agilla [3], DFuse [4], Milan [5], TAG [6], Mires [7]) are usually too general to be effective or demand too many resources to be implemented on sensor node platforms. The third programming approach combines the best characteristics of the other two: it is based on frameworks that include domain-specific libraries and tools that can be easily reused for multiple applications of a selected domain. This approach allows reducing design time through modularity and reuse while offering solutions that are optimized for the target domain. For example, most BSN systems include signal processing intensive tasks. The prototyping of BSN applications can be significantly facilitated by a domain-specific framework with libraries and protocols that allow to implement signal processing tasks efficiently in star-topology networks.

To support the design of optimized BSN applications while minimizing the design time and effort, a domain-specific open source framework, called SPINE (signal processing in-node environment) [8], has been developed. SPINE offers a service architecture and a set of libraries that are common to most BSN applications, and at the same time makes it easy to customize or extend the given libraries to meet the particular needs of specific applications. SPINE provides libraries of protocols, utilities and processing functions and a lightweight Java API that can be used by local and remote applications to manage the sensor nodes or issue service requests. By providing these abstractions and libraries, that are common to most signal processing algorithms used in WSNs for sensor data analysis and classification, SPINE also provides flexibility in the allocation of tasks among the WSN nodes and allows the exploitation of implementation tradeoffs. For example, SPINE supports distributed implementations of classification systems where signal processing functions are computed on the sensor nodes and the result sent to the BSN coordinator (called SPINE coordinator) running on a PC or PDA. This allows reducing the amount of data exchanged between the BSN coordinator and the sensor nodes with respect to applications where sensor nodes transmit raw sensor data.

This paper provides several research contributions.

The first contribution is represented by the SPINE framework, which is able to support effective and rapid development of efficient BSN applications according to a domain-specific framework approach. The SPINE framework is described in details for potential users to understand its architecture and main components and to use it for the rapid development of BSN applications. It is worth noting that the SPINE-based high-level programming and monitoring/control (configuration, initialization, start/restart, etc) of a sensor node and the final data processing is done in Java without requiring any knowledge of the low-level sensor node programming which is currently based on nesC/TinyOS.

The second contribution is the analysis of a SPINE-enabled BSN application (a human activity monitoring system) based on 'in-node signal processing' with respect to a similar application based

DOI: 10.1002/spe

on 'only base station processing' as the majority of applications currently available. In particular, it is shown how the framework can optimize (from software perspective and system performance perspective) the final implementation of the proposed application. Indeed the developed human activity monitoring system is '*per se*' an interesting contribution as it provides high recognition accuracy by using only 2 wearable sensors.

Finally, the third important research contribution is related to the easy reuse of (i) the design schemas and patterns ('software' perspective) and (ii) the high-level communication protocol ('communication' perspective) on which SPINE is based for the development of new BSN applications and for the porting of SPINE itself on other sensor platforms. On such bases, SPINE was also ported to the TI Z-Stack sensor platform [9] with limited efforts and the SPINE coordinator was made interoperable with other hw/sw sensor platforms (e.g. Java-based SunSPOT [10]).

The remainder of this paper is organized as follows. Section 2 outlines some relevant related work on BSN applications and software development techniques. Some BSN applications are presented and the CodeBlue [11] and Titan [12] frameworks are described. Section 3 describes in detail the SPINE framework principles and architecture. The SPINE communication protocol is also discussed. Section 4 presents an efficient implementation of SPINE on the TinyOS operating system [13]. Some code snippets are shown as well as the performance evaluation of SPINE node-side. In Section 5, the base-station side of the SPINE framework is presented. Section 6 describes the case study application developed using SPINE. Moreover, the developed application is compared with the same application implemented without signal in-node processing mechanisms. Finally, Section 7 presents the conclusions and discusses the future research directions.

2. RELATED WORK

Most research efforts on BSNs are currently focusing on proof-of-concept applications with the aim to demonstrate the feasibility of new context-aware algorithms and techniques, e.g. for recognition of physical activity or prompt detection of heart diseases, considering issues such as power consumption and also radio channel usage but not taking into account code reusability and modularity. In [14], a method for physical activity monitoring is presented, which is able to detect body postures and periods of walking in elderly persons using one kinematic sensor attached to the chest. In [15], wearable motion sensors are used to guide post-stroke rehabilitation by models to predict clinical scores of motor abilities. In [16], activity recognition is improved by integrating wearable sensors with ambient blob-based vision sensor data. Another physical activity recognition system based on wearable devices is presented in [17]. In [18] is discussed in detail the interesting issue of developing a personal activity recognition system which is based on data coming from a single body location regardless of the specific sensor location and able to work with different individuals. The system can be further personalized to enhance the activity recognition accuracy. Finally, the most exhaustive effort on activity recognition based on wearable sensors is presented in [19]. All these projects have focused on efficient application-specific solutions rather than on the definition of reusable frameworks facilitating the development of BSN applications.

One of the most relevant attempts to define a general platform able to support various BSN applications is CodeBlue [11]. CodeBlue is a framework running on TinyOS specifically designed for integrating wireless medical sensor nodes and other devices. CodeBlue allows these devices to discover each other, report events, and establish communications. CodeBlue is based on a publish/subscribe-based data routing framework in which sensors publish relevant data to a specific channel and end-user devices subscribe to channels of interest. It includes a naming scheme, a multi-hop communication protocol, authentication and encryption capabilities, location tracking and in-network filtering and aggregation. CodeBlue provides end-user devices with a query interface for retrieving data from previously discovered sensor nodes. Although CodeBlue provides a sensor driver abstraction architecture which allows an easy

DOI: 10.1002/spe

integration of new sensors within the system, selection of sensor types or physical node identifiers as data sources, tuning of the data rate and definition of threshold-based filters to avoid unnecessary data being transmitted, it does not allow inserting complex signal processing functionalities into the sensor nodes. It only supports simple threshold-based triggers on the sensor readings which does not give enough flexibility for the variety of requirements of the BSN applications.

A higher level approach is adopted by Titan [12]. Titan, which is implemented in TinyOS, is a general-purpose middleware that supports implementation and execution of context recognition algorithms in dynamic WSN environments. Titan represents data processing by a data flow from sensors to recognition result. The data are processed by tasks, which implement elementary computations. Tasks and their data flow interconnections define a task network, which runs on the sensor network as a whole. Tasks are mapped onto each sensor node according to the sensors and the processing resources it provides. Titan dynamically reprograms the WSN to exchange context recognition algorithms, handle defective nodes, variations in available processing power, or broken communication links. The architecture of Titan is composed of several software components, which enhance modularity. Although Titan raises the programming abstraction level by offering a middleware for effectively developing signal processing applications in WSNs, it is based on too generic mechanisms for providing efficient solutions in the specific BSN application domain. In fact, the programming of a feature extraction operation on the sensor node, which is often carried out in a BSN application, requires the creation of at least five tasks (sampling, buffering, loading, feature calculation, transmission). Moreover, some overhead can be introduced due to the connections of the output and input ports among tasks through which data are exchanged.

Domain-specific frameworks [8, 20] are in the middle between application-specific code and general-purpose middleware approaches. They specifically address and standardize the core challenges of WSN designs 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, the SPINE framework proposed in this paper is a domain-specific framework in the context of signal processing in-node intensive BSN applications, whereas CodeBlue and Titan can be seen as general-purpose middleware solutions for BSN applications.

To summarize, the development of WBSN applications can mainly be implemented according to the following approaches: (i) application-specific code [14–18], (ii) general-purpose middleware [11, 12] or (iii) domain-specific frameworks [8]. Table I summarizes the characteristics of the above three described approaches. As will be discussed in detail in the remainder of this paper, SPINE allows for both code efficiency similar to an application-specific code approach, and code reusability, rapid prototyping, easy debugging, and system interoperability as CodeBlue and Titan. Moreover, it specifically supports functionalities for flexible sensing operations and easily programmable in-node signal processing.

Table I. Comparison among the available approaches for BSN application development.

	Application-specific code	Domain-specific framework	General-purpose middleware
Code Reusability		√	√
Rapid prototyping		√	√
Ease of debugging		√	√
Code efficiency	√	√	
System interoperability		√	√
Specific support to flexible sensing operations		√	
Specific support to in-node signal processing		√	

DOI: 10.1002/spe

3. THE SPINE FRAMEWORK

SPINE is a domain-specific framework for the development of BSN applications which is based on the following principles:

- *Open Source*. SPINE is an open source project to establish a broad community of users and developers. Its source code is released under the LGPL 2.1 license [21].
- *Interoperability through APIs*. SPINE provides local and remote applications with lightweight Java APIs that can be used to manage sensor nodes or issue service requests. The APIs are easily portable to devices of various capabilities, such as PCs or mobile phones.
- *High-level abstractions*. SPINE provides high-level programming libraries including protocols, utilities, data processing functions and specific support to easily specify new services and features. The layer defined by the SPINE service libraries allows application designers to program at higher levels of abstraction than the currently available operating environments (e.g. TinyOS).
- *Rapid prototyping*. SPINE helps designers to efficiently prototype distributed BSN data classification algorithms with respect to the use of energy and channel bandwidth.

Table II introduces the most common task/processes involved in BSN applications on sensor nodes, which are supported by SPINE.

In the following subsections, the SPINE network architecture, the functional architecture and the application-level communication protocol between base station and sensor nodes are described.

3.1. Network architecture

The BSN architecture supported by SPINE includes multiple sensor nodes and one coordinator node (see Figure 1). The coordinator manages the network, collects and analyzes the data received from the sensor nodes, and acts as a gateway to connect the BSN with wide area networks for remote data access. Sensor nodes measure local physical parameters and send raw or processed data to the coordinator. Currently, SPINE supports BSNs with star topology, which is a requirement for BSN applications, where sensor nodes communicate only with the coordinator. However, the framework can be easily extended to also support direct and multi-hop communications among sensor nodes. In the current version of SPINE (version 1.3) a sensor node can only be associated

Table II. Common tasks supported by BSN applications at node-side.

TASK	DESCRIPTION
SAMPLING	The sensor sampling process represents the first step for developing a BSN application. Selecting the appropriate sampling time to satisfy the application requirements is important because the amount of data generated and processed, and under certain degree the energy consumed depend on it.
FEATURE EXTRACTION	Classifier algorithms very rarely use raw data. Instead, attributes (or features) are typically extracted on data windows and used to detect events and classify activities. Extracting features directly on the wireless nodes also allows the reduction of the radio usage, as aggregated results are sent instead of several raw data values.
QUERIES	Support for selective queries on the available sensors of a node is important because application requirements can change over time and not all the sensors are necessarily involved for algorithms execution at any time.
NODE SYNCHRONIZATION	In a WBSN, nodes should be kept synchronized when sampling the sensors and processing data, because data gathered from multiple wearable nodes must refer to the same time interval to be aggregated to recognize correctly e.g. physical activities or other events of interest.
DUTY CYCLING	Duty cycling is a mechanism for handling the radio status (idle, on, off) to reduce power consumption of a sensor node and therefore its battery lifetime. In particular, radio duty cycling must be tuned very carefully, reducing as much as possible the active time (transmitting, receiving, and listening).

DOI: 10.1002/spe

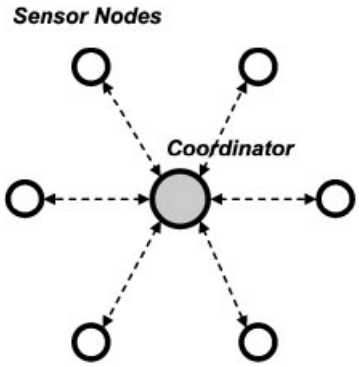


Figure 1. SPINE network architecture.

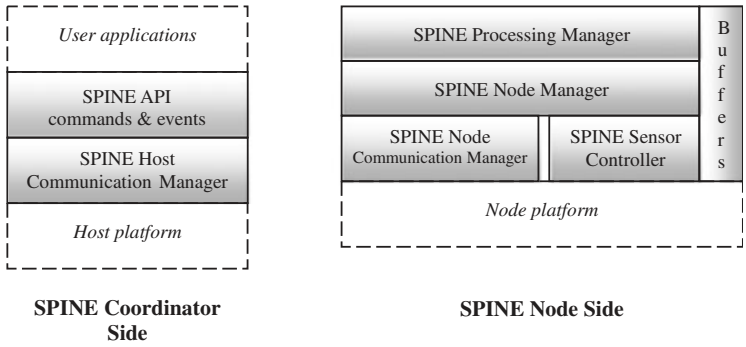


Figure 2. SPINE functional architecture.

with a single coordinator; a possible extension is to allow nodes to be associated and communicate with multiple coordinators. A scenario where such architecture could be used is when a patient wearing body sensors moves across locations; in this case such sensors should connect to a different coordinator at each different location.

3.2. Functional architecture

The SPINE framework consists of two main components, implemented, respectively, on the coordinator (e.g. a PC or a smart-phone) and on the BSN nodes. The functional architecture of SPINE is shown in Figure 2.

On the coordinator side, SPINE provides application developers with a very intuitive interface to the BSN which is placed between end-user applications and the hardware and software host platform. User applications manage a SPINE-based network through the lightweight and well-defined *SPINE API*. The surface level of SPINE lets the registered applications be notified of high-level events generated by the remote BSN, such as discovery of new nodes, sensor data communication, node alarms, and system messages such as low battery warnings. Commands issued by the user application and network-generated events are respectively coded in lower level SPINE messages (see Section 3.4) and decoded in higher level information by the *SPINE Host Communication Manager*. This component takes care of packets generation and retrieval and interfaces with the specific software components of the host platform to access the physical radio module to transmit/receive packets to/from the BSN.

On the node side, the SPINE framework is responsible for providing developers with abstractions of hardware resources such as sensors and the radio, a default set of ready-to-use common signal processing functions and, most important, a flexible and modular architecture to customize and extend the framework itself to support new physical platforms and sensors and introduce new signal processing services. In particular, the *SPINE Node Communication Manager* acts as the

DOI: 10.1002/spe

counterpart of the SPINE Host Communication Manager; in addition, it possibly takes into account management policies to optimize energy consumption by an intelligent use of the radio module (see Section 3.3). The *SPINE Sensor Controller* manages and abstracts the sensors on the node platform, providing a standard interface to the diverse sensor drivers. It is responsible for sampling the sensors and storing the sensed data in properly defined *Buffers*. The *SPINE Node Manager* is the central component, responsible for recognizing the remote requests and dispatching them to the proper components. Finally, the *SPINE Processing Manager* consists of a dispatcher for the actual processing services and a standard interface for user-defined services integration.

At compile time, SPINE allows developers to tune a number of parameters of the sensor node. In particular, it is possible to specify which sensor drivers and processing functions must be included in the compilation, how many buffers must be allocated for the sensors, and what size such buffers must have. Other compile time parameters (e.g. radio channel, max active message payload size) are tunable as in any other TinyOS application.

At run-time, SPINE allows to tune several parameters of the sensor nodes and activate and deactivate functions and alarms. First of all, each available sensor of a remote node is initially idle, but can be set to start sampling at any time. Sampling time and time scale (e.g. 'ms', 's', 'min') are tunable parameters. Second, any computable processing function (e.g. feature extractors or alarms) is active by defaults, but they can be started or stopped remotely. Typical tunable processing parameters include window of samples, shift over the window, type of feature extractors to compute on a given sensor, type and value of a threshold that would trigger alarms on given sensed data. It is also possible to enable/disable a simple Time Division Multiple Access (TDMA) communication protocol, and a 'radio low power' mode through a duty cycle mechanism (see next section for further details). SPINE also introduces an optional encryption service that enables the remote motes to communicate securely with the base-station node. This service is currently available for CC2420-based platforms such as the Telosb/tmote sky and the Shimmer, as it uses the built-in AES-128 hardware encryption.

3.3. The SPINE application-level communication protocol

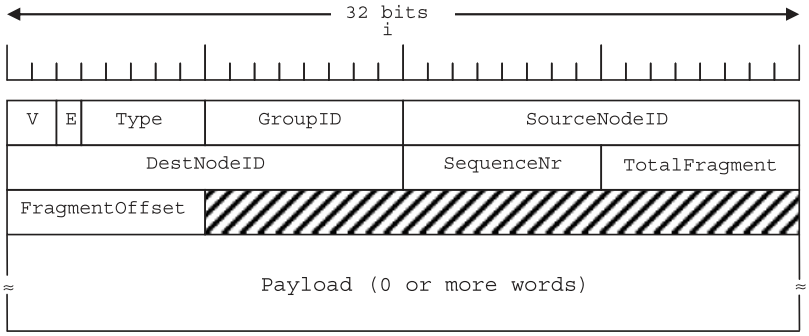
The SPINE framework includes an application-level communication protocol to manage the bidirectional communication between nodes and the coordinator. The SPINE communication protocol works at the application-level and is independent of the underlying network and data-link layers. The current architecture includes an optional TDMA scheme and a radio duty cycling mechanism.

A general communication scheme has been defined and is supported by a set of standard messages summarized in Table III, whereas the structure of the general SPINE message is reported in Figure 3. Messages can be directed from the coordinator to a node (C→N) or from a node to the coordinator (N→C). While service messages have a fixed format, user messages can be easily customized to better fit the application needs. Moreover, developers also have the possibility to extend the framework with new user-defined messages.

Table III. Standard messages of the SPINE protocol exchanged between Coordinator (C) and Node (N).

	Direction		
	C→N	N→C	Parameters
Service Discovery	•		NONE
Service Advertisement		•	< sensors list, services list>
Set-Up Sensor	•		< sensor code, sensor parameters >
Set-Up Service	•		< service code, service parameters>
Activate Service	•		< service code >
De-activate Service	•		< service code >
Data (raw or processed)		•	< service code, data >
Start processing	•		< radio configuration >
Reset (node/network)	•		NONE
System notification		•	< notification type, notification details >

DOI: 10.1002/spe



V = Version of the protocol
E = Extension flag signalling an extension of the message
Type = Type of SPINE message
GroupID = Identifier of the group
SourceNodeID = Identifier of the transmitting node
DestNodeID = Identifier of the receiving node
SequenceNr = Sequence number of the messages
Total Fragments = Number of total fragments of the SPINE message
Fragment Offset = Offset of the fragment with respect to the reference message

Figure 3. Structure of the general SPINE message.

The communication scheme is usually initiated by a *Node/Service Discovery* phase. Several discovery techniques can be adopted. For example, the coordinator can broadcast a service request (Service Discovery message) and wait for a service advertisement by active nodes within range. To keep the network information updated, service requests can be sent on a periodical basis. A Service Advertisement, sent from a sensor node upon reception of a Service Discovery message, includes information about the node hardware, in particular regarding the available sensors, and the node services which can be processing functions (e.g. mean, max, min, variance, total energy, entropy) and/or alarms (sensed data exceeding thresholds). Table IV reports the services and alarms that SPINE provides by default. It is worth noting that many services can be set up to process periodically, on-demand only, or even on event basis. Hence, the data flow generated by the service is dependent on the service itself and on the specific dynamic configuration requested by the application on that service. For instance, a raw data service can be set up to transmit sensor data periodically, one-shot, or only when a given threshold has been exceeded.

According to the information collected, the *User application* on the coordinator must first set up the sensors of interest by specifying: sensor identification code, sampling time, and time scale. Once sensors have been set-up, the User application will typically set-up the desired services (if the service requires a preliminary set up phase) and subsequently activate them. Services can be dynamically activated and, if necessary, de-activation is also supported.

Set up, activation and de-activation of a service involve the specification of certain parameters which usually vary from service to service. Thus, corresponding messages have been structured with a dynamic and partly service-specific format. As aforementioned, the SPINE framework provides native services, whereas new services can be easily integrated. In this case, the developer must enhance the framework with a specific format for the set-up, activation and de-activation request messages.

Once each node is fully set up, the user application is finally ready to start working by broadcasting a start message. From this time on, the nodes will start sensing and processing according to the activated services. The data produced by the running services of each node are transmitted in data messages to the coordinator which in turn forwards the message content to the user application. The data message has a well-defined format, but its payload semantic is clearly dependent on the service that generated that data and, again, if it is a user-defined service, coding and decoding of the payload content is up to the developer. To notify the coordinator of system events such as

DOI: 10.1002/spe

Table IV. Default services and alarms which can be activated in the SPINE node.

FEATURE	DESCRIPTION
Raw data	Sensed data coming from sensing processes
Max	Maximum value computed on a sample window
Min	Minimum value computed on a sample window
Range	Maximum displacement (max–min) value computed on a sample window
Mean	Average value computed on a sample window
Amplitude	(Maximum-mean) value computed on a sample window
RMS	RMS value computed on a sample window
St dev	Standard deviation value computed on a sample window
Total energy	Cross-axial magnitude computed on a sample window. It takes into account multiple sensor channels, if any.
Variance	Variance value computed on a sample window
Mode	Most frequent value computed on a sample window
Median	Median value computed on a sample window (central value of the ordered window buffer)
Vector magnitude	Magnitude of a sample window (sum of the squares of the window elements)
Entropy	Entropy computed on a sample window
Pitch & roll	Pitch and roll estimation computed on a sample window. It is useful only if applied to accelerometer data.
ALARM	
ABOVE	An alarm is triggered when a given sensor data or a computed feature exceeds the specified threshold
BELOW	An alarm is triggered when a given sensor data or a computed feature goes below the specified threshold
WITHIN	An alarm is triggered when a given sensor data or a computed feature are within the range of the specified thresholds (min, max)
OUTSIDE	Alarm is triggered when a given sensor data or a computed feature exceeds the range specified by the thresholds (min, max)

low battery warnings, processing overhead errors or bad requests, nodes can issue defined system event notification messages. Single nodes or the whole network can be reset by the coordinator if requested by the user application. Information such as sensor and service advertisements, error and warning types and details, are all exchanged in the form of numerical codes which must be shared between the nodes and the coordinator.

To provide a concrete example of a typical messages exchange between the coordinator and a node, Figure 4 shows the sequence of transmissions within a scenario where a user application is looking for a node equipped with a given sensor to request some in-node processing of the data sensed by that sensor. The user application, through the coordinator side of SPINE, broadcasts a Service Discovery to check if a node with the required sensor is found in the vicinity. An active node replies to the service discovery with a Service Advertisement. The information contained in the advertisement message is sufficient for the user application to understand whether that node has the sensor and the signal processing services required. In that event, the application can proceed by first setting up the sensor with the desired sampling rate. Then, the application will set up the specific function(s) and subsequently activate them. The node configuration phase is now complete and the application issues a start message which includes configuration for radio behaviors of the sensor nodes (enabling/disabling of TDMA, number of nodes for TDMA initialization, and radio activity of type *always on* or *duty cycle*). Upon reception of the start message, the node reacts by starting the sampling timer on the sensor and the processing mechanism, and transmitting the results as they are available.

An important mechanism of SPINE is the radio duty cycle, a simple run-time mechanism to help in reducing the power consumption due to the radio usage. When enabled by the coordinator, the sensor nodes turn on the radio only when they need to transmit data over the air. In order to receive messages, before turning off the radio after the successful transmission of a message, the radio is kept on listening to incoming messages for a given period of time (duty cycling timer),

DOI: 10.1002/spe

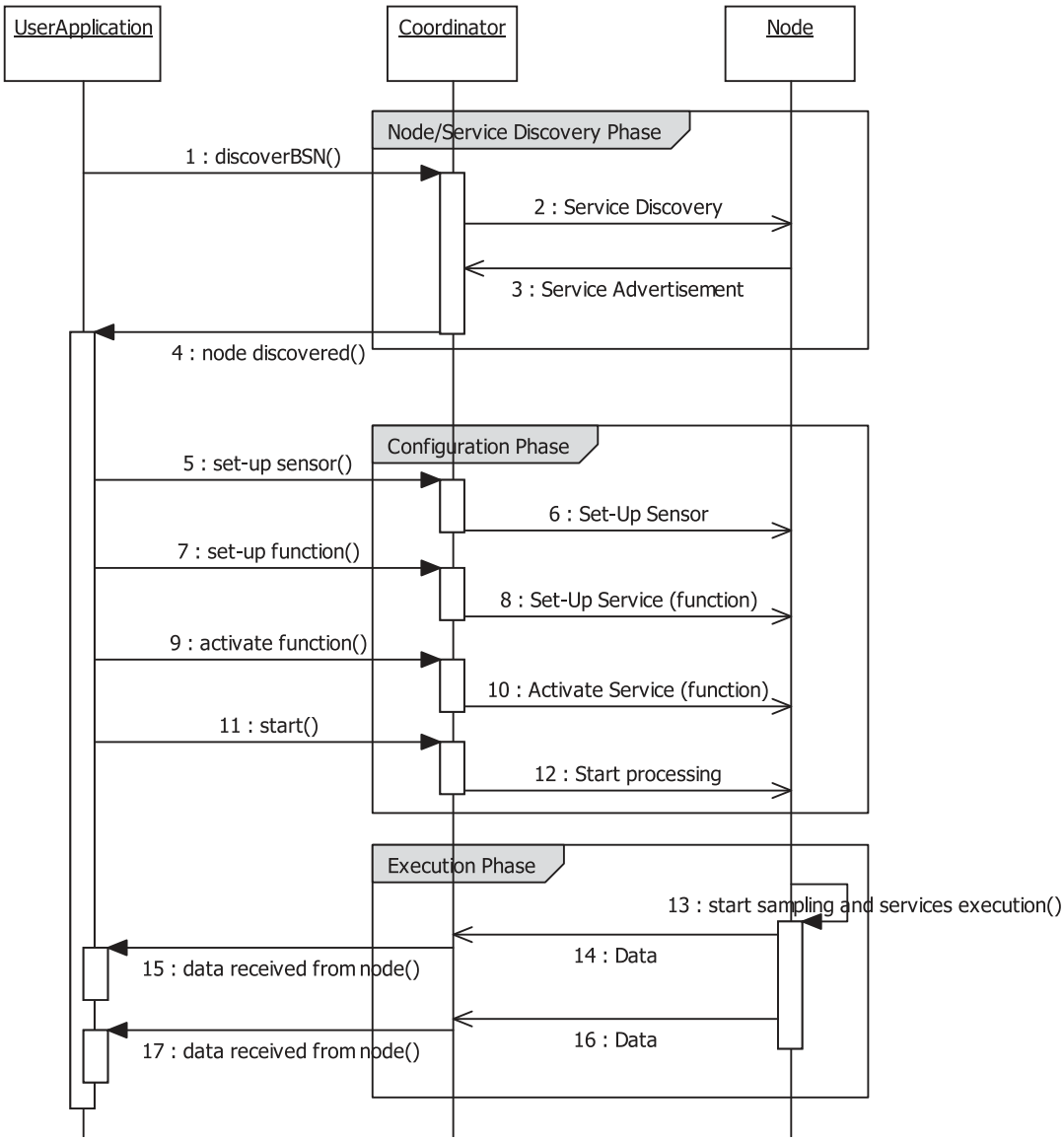


Figure 4. Example of communication between user application, coordinator and sensor node.

which can be set at compile time (usually in the order of a few milliseconds). If a message is received the timer is reset. The coordinator, which keeps a queue of the messages to be sent to each node, sequentially sends each message for a given node immediately after receiving a packet. If an ack is received for a sent message, the coordinator removes that message from the queue of messages ready to be sent.

4. THE TINYOS-BASED NODE SIDE OF SPINE

In this section the node side of the SPINE functional architecture presented in Section 3.2 is described by (i) detailing the developed TinyOS-based architecture, (ii) presenting some programming examples on how to use and easily extend the framework and (iii) discussing the obtained performance evaluation results.

DOI: 10.1002/spe

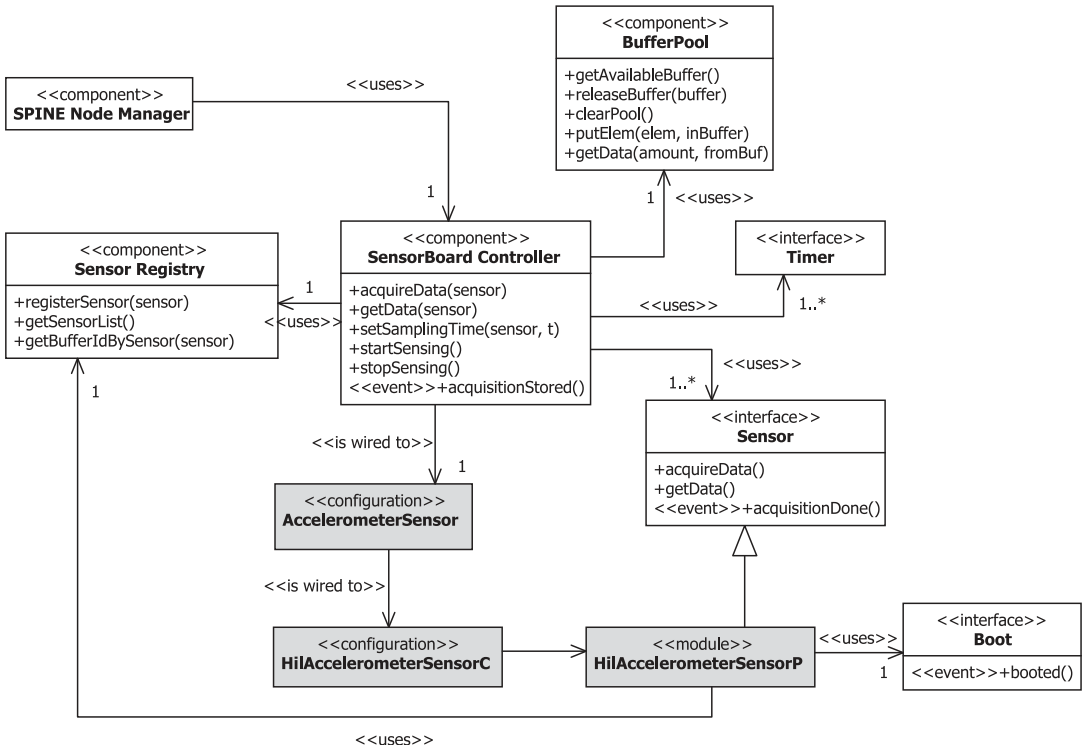


Figure 5. Class diagram of the sensing logical block.

4.1. *Software architecture in TinyOS*

The flexible and effective TinyOS software architecture of SPINE is presented in the following. The presentation of the static architecture is aided by UML class diagrams, whereas UML sequence diagrams are used to show key dynamic interactions among software components. The whole architecture has been graphically shown in three separate class diagrams (see Figures 4, 6, and 8). Logical components are reported as single blocks where not strictly necessary. The stereotype *<<component>>* has been used to represent TinyOS entities which typically consist of three components: an *interface* for publishing the component commands and events, a *configuration* for wiring external components and a *module* which implements the component commands and that can generate the defined events of the component interface. The stereotype *<<is wired to>>* indicates the TinyOS wiring operation '→' [13].

4.1.1. Sensing. The class diagram in Figure 5 shows the architecture of the SPINE Sensor Controller (or sensing functional block). To enhance extendibility, access to the sensors drivers has been decoupled by the introduction of the SensorBoard Controller and the Sensor interface. Thus, sensors are addressed only by unique codes. In particular, by using parameterized Sensor interfaces, the SensorBoard Controller module is itself independent of the actual sensors; wiring the actual sensor drivers (i.e. HilAccelerometerSensorC) to the parameterized Sensor interface is left to the SensorBoard Controller configuration component, which is much easier to edit or extend. Each sensor driver module (i.e. HilAccelerometerSensorP) must register its unique sensor code to the Sensor Registry to inform (through the Service Advertisement message) the coordinator of the presence of the sensor. The SensorBoard Controller module uses timers for the sampling operations of the available sensors.

In TinyOS sensor data are typically gathered in 'split-phase', which means an operation request and its correlated response are separate with a callback mechanism. Hence, as shown in the sequence diagram of Figure 6, when a given sensor timer fires, the SensorBoard Controller requests the

DOI: 10.1002/spe

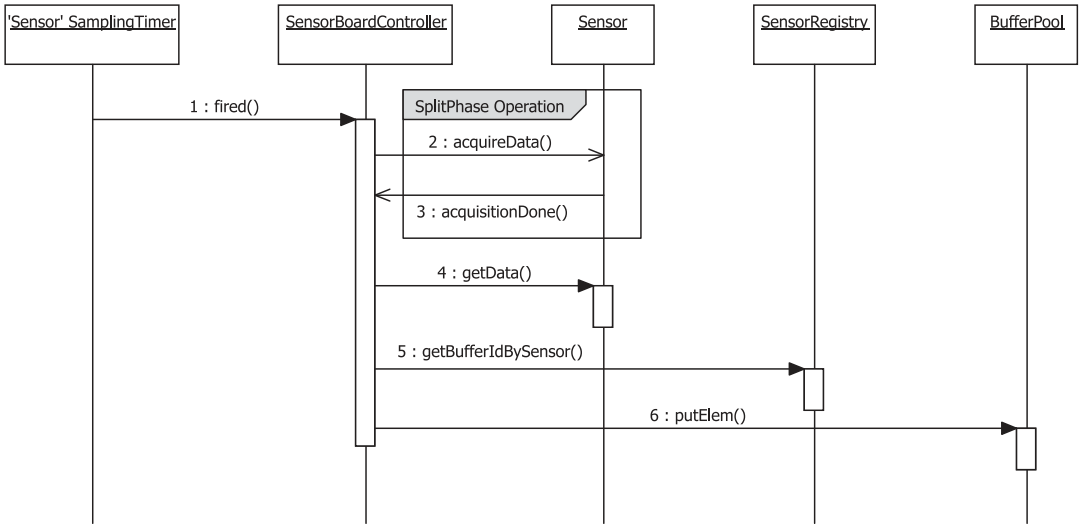


Figure 6. Sequence diagram of the sensing process.

corresponding sensor to start data acquisition. When the data are ready, the sensor driver notifies the SensorBoard Controller which, in turn, can get the new reading. Finally, to decouple data sources (the sensors) by data consumers (the processing services), sensor readings are stored by the SensorBoard Controller in an *ad hoc* BufferPool. The BufferPool is internally implemented as a set of circular buffers. Mappings between buffers and sensors are stored in the Sensor Registry.

4.1.2. Processing. The SPINE Processing Manager (or processing block) (see Figure 7) relies on a similar schema adopted for the sensing block to support fast extensions of the processing functions provided by the framework. In fact, the Function Manager handles the actual implemented functions through the parameterized interface Function. Features are particular types of functions which are applied on windows of sensed data. In particular, they are characterized by the following parameters: *Window*, which is the number of buffered data samples on which the function is applied, and *Sliding%*, which is the percentage of shift on the buffered data samples with respect to the *Window*. For instance, if *Window* = 40 and *Sliding%* = 50, the function is computed on 40 acquired samples composed of the last 20 previously acquired samples and the first 20 newly acquired samples.

A specific type of function is the FeatureEngine which handles particular math functions named features (e.g. max, min, standard deviation, etc.). In particular, the FeatureEngine acts as a dispatcher for accessing the various feature extractor components and as an aggregator if multiple features are requested to be computed on the same data. Functions are invoked through the Function Manager by unique codes. Besides centralizing the access to the various functions, the Function Manager mainly provides a data transmission command which masks the presence of lower level transmission services to the functions. This choice is motivated by standardization issues of data messages, to guarantee the creation of standard messages, function data are encapsulated by the Function Manager as the payload of a new SPINE data message. It should be noted that the Function interface is strictly part of the framework core and is used to generalize the concept of processing function, such as feature extractors, alarms, sensor data filters and pre-processors, and even simple online classifiers. The Feature interface, instead, has the sole scope of decoupling the actual feature extractors from the FeatureEngine. Section 4.2 explains such differences, by providing two examples on how to add a new feature extractor and a new processing function.

Figure 8 shows the complete sequence of steps from buffer data fetching to features extraction computation. As aforementioned, processing functions are completely decoupled from the sensor data generation; to get the proper data frames, a function, as soon as a new computation is required, accesses the Sensor Registry to get the buffer *id* associated to the sensor of interest; then, it

DOI: 10.1002/spe

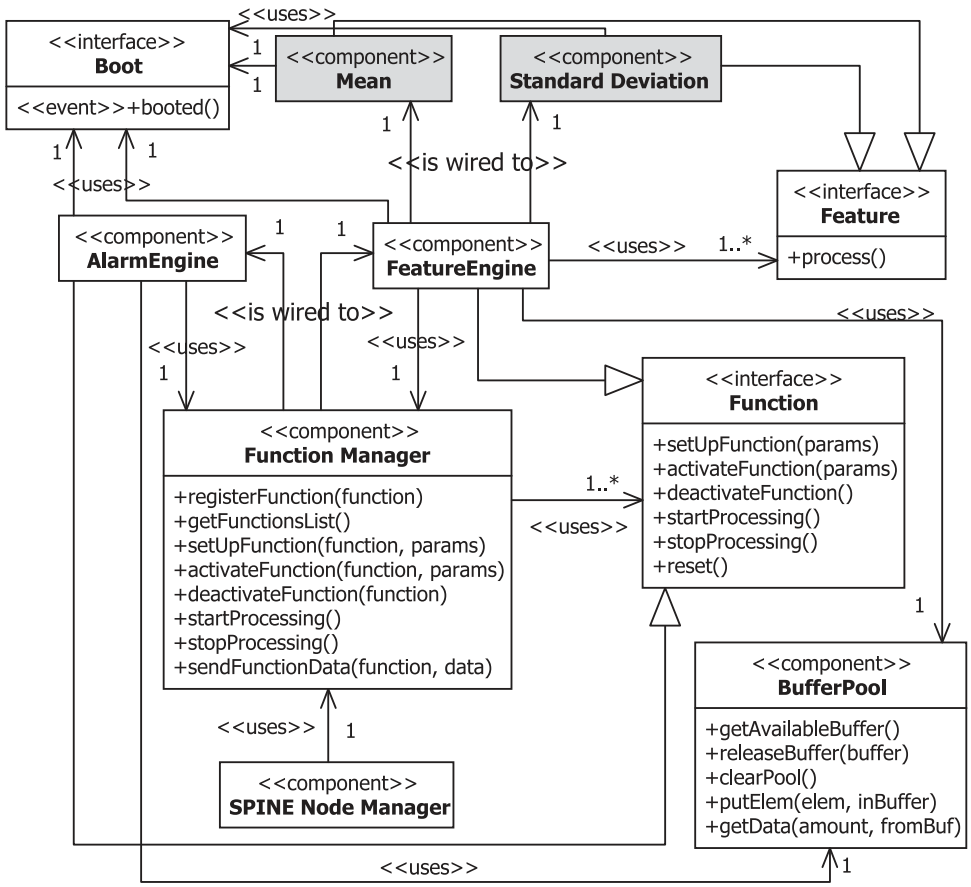


Figure 7. Class diagram of the processing logical block.

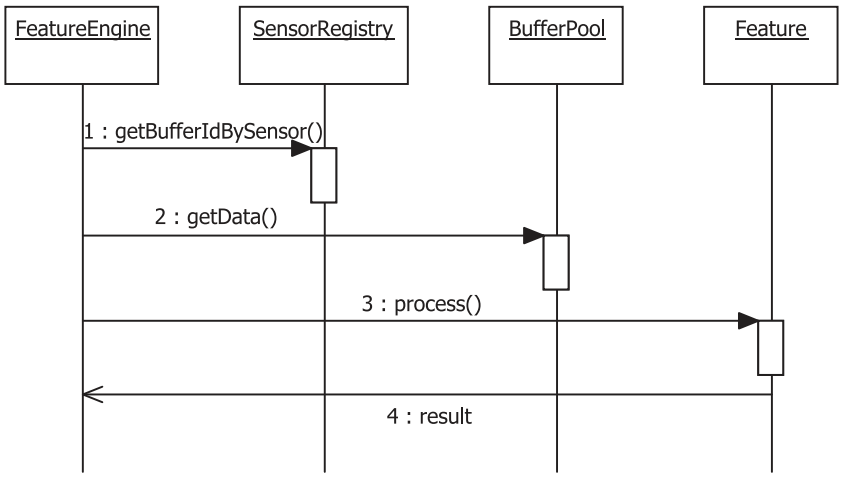


Figure 8. Sequence diagram of a feature processing.

obtains the desired data amount on that buffer through the BufferPool component and computes the processing.

4.1.3. Communication. The SPINE Node Communication Manager (or communication block) is presented in Figure 9 to explain how messages transmission and reception are handled at the

DOI: 10.1002/spe

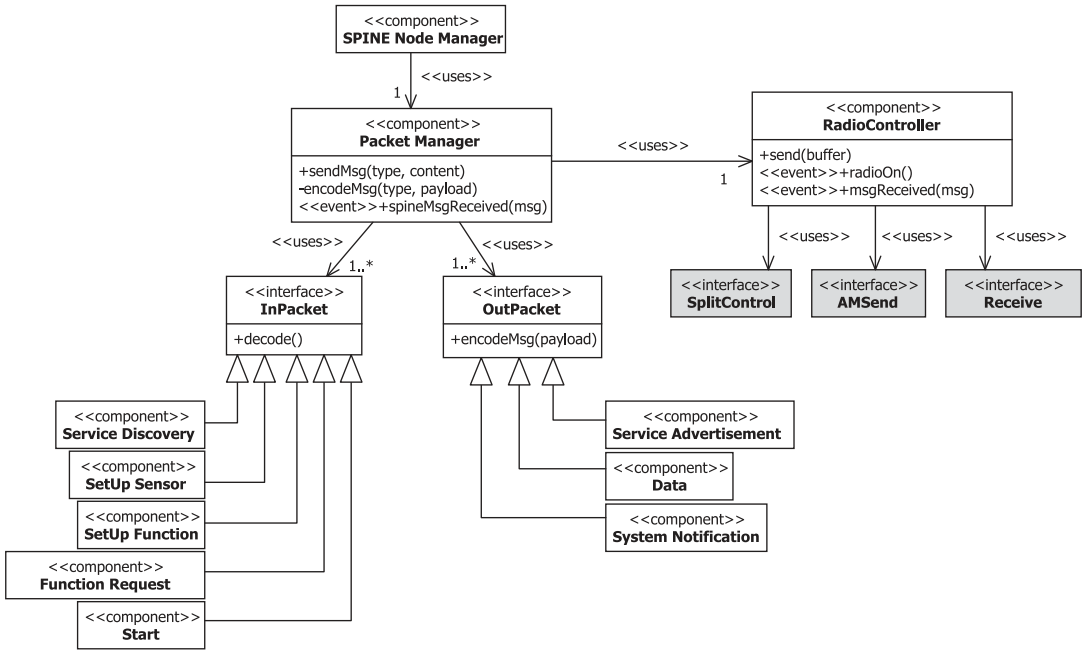


Figure 9. Class diagram of the communication logical block.

SPINE node side. At the lowest level, the RadioController masks the TinyOS components for controlling the physical radio states (e.g. turn on/switch off) and handling packets transmission and reception. The RadioController exposes the *send* command to transmit a stream of bytes, which are not further manipulated. Users of the RadioController are not aware of the current state of the radio module, which is transparently managed by the RadioController. Additionally, the RadioController signals the event of the reception of a new packet. Such events are captured by the Packet Manager which verifies whether the received packet is a SPINE message and, if that is the case, generates a new *spineMsgReceived* event. SPINE messages are decoded and encoded by different components one for each message type defined (see Section 3.3). The Packet Manager, after having recognized the type, which is contained into the SPINE header of each message, dispatches the encoding (if it is an outgoing message) or the decoding (if it is an incoming message) to the proper codec component. The dispatching is implemented through parameterized interfaces: InPacket for incoming messages and OutPacket for outgoing messages.

The sequence diagram in Figure 10 shows how the reception of a new message is captured and processed by SPINE. In the example, a user application sends over-the-air a Set-Up Sensor request. The packet is received by the RadioController which forwards it 'as is' to the Packet Manager. The Packet Manager checks, by processing the packet header, that it is a valid SPINE message and requests the message decoding to the proper decoder; then it generates a *spineMsgReceived* event, notifying the SPINE Node Manager with the message type. The SPINE Node Manager handles this event by invoking an internal procedure to process the message parameters coming from the SetUp Sensor decoder component. Finally, it invokes the SensorBoard Controller to set up the sampling timer needed to drive the sensing operation of the given sensor.

An example of message transmission is shown in Figure 11. After having computed the activated features, the FeatureEngine sends back the results by invoking the *sendFunctionData* command of the Function Manager; the Function Manager generates a new data transmission request to the Packet Manager which, in turn, encodes this request into a Data message and invokes the Radio Controller to transmit it over-the-air.

DOI: 10.1002/spe

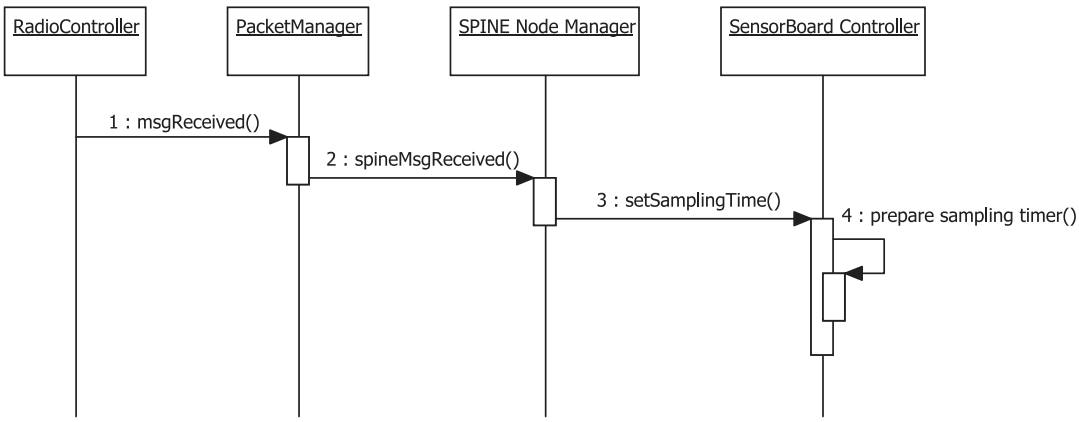


Figure 10. Sequence diagram of a message reception and handling.

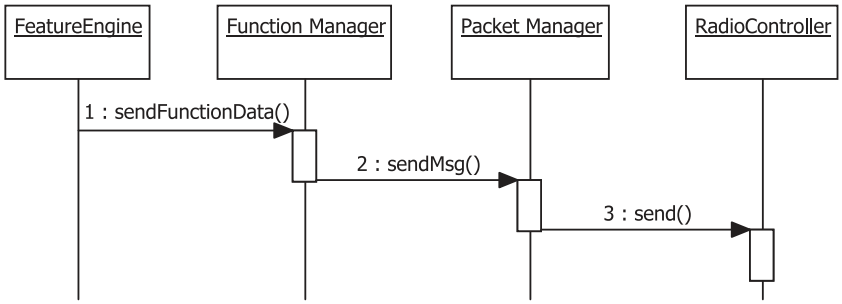


Figure 11. Sequence diagram of a data message transmission.

4.2. Programming examples

Some programming examples on how to use and extend the node side of the SPINE framework are described in the following. Common practices that developers may incur while programming on SPINE are:

- introduction of a new feature extractor;
- introduction of a new processing function;
- integration of new sensor drivers.

In fact, although the SPINE implementation for TinyOS comes with predefined commonly used processing functions, such as feature extractors on sensor data (e.g. mean, standard deviation, RMS, max, min) or threshold-based alarms, and has native support for a set of sensor platforms and sensor boards (Telosb with the SPINE sensor-board and the bio-sensor board [22], MicaZ [23] with mts300 sensor board, Shimmer platform [24]), application-specific processing, as well as special-purpose sensors, may frequently need to be introduced. The peculiar design of SPINE allows for a fast familiarization with the framework and a very flexible extension mechanism, particularly focused on processing and sensing extensions. Customizing SPINE does not require any modification to the core components but only some interconnection code lines to make the new components work.

4.2.1. Introduction of a new feature extractor. Extending SPINE with new features is straightforward. It first requires declaring a new constant in the features enumeration structure, located in a SPINE configuration file. This simply implies a new line for the new feature code (of course not

DOI: 10.1002/spe

yet in use):

```
ENTROPY = 15
```

The following example illustrates the introduction of the Entropy feature, which can be considered as a new feature extractor on sensed data. The Entropy feature component must then provide (implement) the Feature interface and hence the *process* command (see Figure 6) which will contain the feature implementation logic.

The entropy function, commonly used in information theory, is a measure of the (im)purity of an arbitrary collection of examples. In the context of signal processing its value is used to have some kind of information about how sampling data change over the time. In fact, considering an array of samples, an entropy value that equals to 0 means that every data of the array has the same value. Instead, higher entropy values correspond to the fact that the array contains samples having different values. The entropy function is also commonly adopted for attributes selection in a decision tree [25]. The entropy formula is reported in the following:

$$E(X) = -\sum_{i=1}^{n} (p(x_i) * \log(p(x_i))$$

where $p(x_i)$ is the probability of the element/data value x_i considering array X.

The value of $E(X)$ varies between 0 and 1. The SPINE implementation of the function above returns an integer value, considering E(X) multiplied by 1000.

The code of the Entropy feature is reported below.

```
module EntropyP {
 provides interface Feature;
 uses interface Boot;
 uses interface FeatureEngine;
}

implementation {
 bool registered = FALSE;
 event void Boot.booted() {
  if (!registered) {
   call FeatureEngine.registerFeature(ENTROPY);
   registered = TRUE;
  }
 }
 command uint8_t Feature.process(int16_t** data, uint8_t channelMask,
                    uint16_t dataLen, int8_t* result) {
  uint8_t k;
  uint8_t mask = 0x08;
  uint8_t rChCount = 0;
  uint16_t i=0, j=0, diffCounter= 0;
  uint16_t diffValues[dataLen];
  float prob[dataLen];
  float entropy= 0;
  for (k = 0; k<MAX_VALUE_TYPES; k++)
   if ( (channelMask & (mask>>k)) == (mask>>k)){
    for(i=0; i<dataLen; i++){
     for(j=0; j<diffCounter; j++)
      if(data[k][i]==diffValues[j]) break;
```


DOI: 10.1002/spe

```
     if(j==diffCounter){
      diffCounter++;
      diffValues[j]= data[k][i];
      prob[j]= 1;
     }else prob[j]++;
    }
    for(i=0; i<diffCounter; i++) prob[i]= prob[i]/dataLen;
    for(i=0; i<diffCounter; i++)
     entropy+= prob[i]*logf(prob[i]);
    ((uint16_t *) result)[rChCount++] = (uint16_t)(entropy*-1000);
   }
  return channelMask;
 }
 command uint8_t Feature.getResultSize() {
  return 2;   // uint16_t = 2bytes
 }
}
```

Furthermore, it is necessary to notify the system about the existence of the Entropy feature by self-registering the code at boot time. That is done by using the TinyOS Boot interface and hence registering the code in the 'booted' event body (see the method `event void Boot.booted()` in the implementation of the Entropy feature).

The last step consists in wiring the Entropy feature to the FeatureEngine, simply by locating the FeatureEngine configuration component, declaring the Entropy Feature and wiring it to the parameterized Feature interface, as shown below.

```
components EntropyC;
FeatureEngineP.Features[ENTROPY] -> EntropyC;
```

4.2.2. Introduction of a new processing function. The example illustrates the introduction of a new processing function specifically designed for on-node human steps counting.

First of all, a new constant must be included in the functions enumeration, located in a SPINE configuration file, as shown below.

```
STEP_COUNTER = 5
```

The StepCounter component must provide the SPINE interface Function. Finally, it must use the TinyOS Boot interface which implies the implementation of the 'booted' event. Self-registration code must be inserted here, as shown below.

```
event void Boot.booted() {
 FunctionManager.registerFunction(STEP_COUNTER);
}
```

To obtain sensor data, different approaches are available and selecting the best one depends on the nature of the processing function. The more general and, sometimes, necessary way is to use the SPINE interface BufferPool within the StepCounter component. This allows (i) to be notified of new data available in the buffers (specified per IDs) and (ii) to get a single element or data window from the buffers. Such operations require the use of the SensorRegistry to have a reference resolution from buffer identifier to associated sensor.

DOI: 10.1002/spe

However, a simplified StepCounter logic enables a faster approach. The StepCounter uses the SensorBoardController to be notified of new sensors acquisitions, discarding readings coming from any sensors but the accelerometer. In particular, each time the accelerometer is sampled, the x and z axis values are compared with two, experimentally estimated thresholds. If the threshold conditions are satisfied, a proper message is sent over-the-air and a wait counter is set. The purpose of that counter is to avoid multiple recognition of the same step. The complete StepCounter module is reported below.

```
module StepCounterP {
 provides interface Function;
 uses {
  interface Boot;
  interface FunctionManager;
  interface SensorBoardController;
 }
}
implementation {
 bool active = FALSE, start = FALSE;
 uint8_t waitCounter = 0;
 uint16_t steps = 0;

 event void Boot.booted() {
  FunctionManager.registerFunction(STEP_COUNTER);
 }
 command bool Function.setUpFunction(uint8_t* functionParams,
                 uint8_t functionParamsSize) {
  return TRUE;
 }
 command bool Function.activateFunction(uint8_t* functionParams,
                      uint8_t functionParamsSize) {
  active = TRUE;
  return TRUE;
 }
 command bool Function.deactivateFunction(uint8_t* functionParams,
                      uint8_t functionParamsSize) {
  active = FALSE;
  return TRUE;
 }
 command void Function.startProcessing() { start = TRUE; }
 command void Function.stopProcessing() { start = FALSE; }
 command void Function.reset() { start = FALSE; active = FALSE; }
 event void SensorBoardController.acquisitionStored
   (enum SensorCode sensorCode, error_t result, int8_t resultCode) {
  int32_t x, z = 0;
  uint8_t msg[2];
  if(activated && started) {
   if (sensorCode == ACC_SENSOR) {
    if (waitCounter == 0) {
     x = call SensorBoardController.getValue(ACC_SENSOR, CH_1);
     z = call SensorBoardController.getValue(ACC_SENSOR, CH_3);
     if (x < X_THRESHOLD_THIGH && z > Z_THRESHOLD_THIGH){
      waitCounter = DEFAULT_WAIT;
      steps++;
```


DOI: 10.1002/spe

```
        msg[0] = steps;
        call FunctionManager.send(STEP_COUNTER, msg);
       }
      }
      else waitCounter--;
     }
    }
   }
}
```

A further observation is related to the ease of transmitting data using the communication functionalities provided by SPINE compared to the usual way to send over-the-air messages in TinyOS. In the previous example, a single line of code is necessary:

```
call FunctionManager.sendFunctionData(STEP_COUNTER, data);
```

Finally, the StepCounter must be wired to the FunctionManager to be accessed remotely. This is done by adding the declaration of the StepCounter component and wiring it to the parameterized interface Function in the FunctionManager configuration.

```
components StepCounterC;
FunctionManagerP.Functions[STEP_COUNTER] ->StepCounterC;
```

It is important to note that only two modifications are necessary to the framework components (to a configuration file for adding the new function constant and to the FunctionManager configuration component to wire the new function). Such modifications can be very easily made by automatic code generation aided by a simple and friendly graphical user interface. This tool could enhance the SPINE flexibility even more.

4.2.3. Introduction of a new sensor. The following example shows how to integrate a new sensor in SPINE. SPINE has been designed to introduce an abstraction level for sensor drivers. Thus, it may happen that the framework already offers support for the type of sensor the developer must integrate, but probably not for the specific hardware component. For instance, SPINE supports accelerometer sensors, but only the drivers for an *ST* accelerometer have been already implemented. Many other times, however, a completely new sensor could be involved. As the former situation is a sub-case of the latter, the focus now is on the introduction of an electrocardiogram (ECG) sensor. First of all, a proper configuration file must be updated by declaring the new sensor type (nevertheless the specific hardware chip brand/model) into the sensors enumeration structure, as shown below.

```
ECG_SENSOR = 7
```

Then, a new configuration component representing the sensor abstraction level must be implemented, as shown below.

```
configuration ECGSensorC {
 provides interface Sensor;
}
implementation {
 components HilECGSensorC;
 Sensor = HilECGSensorC;
}
```


DOI: 10.1002/spe

The purpose of this component is to allow fast updates if, for instance, an old ECG sensor is replaced by a new one, e.g. more energy efficient.

In fact, the framework provides this high-level of abstraction rather than dealing with the specific sensor drivers. Hence, the SensorBoardController must be updated as well, by (i) declaring the new sensor driver component and wiring the parameterized Sensor interface to it and (ii) declaring a new sampling timer to be wired to the parameterized interface Timer, as shown below.

```
components ECGSensorC;
components new TimerMilliC() as ECGTimer;
SensorBoardControllerP.SensorImpls[ECG_SENSOR] -> ECGSensorC;
SensorBoardControllerP.SamplingTimers[ECG_SENSOR] -> ECGTimer;
```

Finally, the actual driver for the physical ECG sensor available must be implemented providing the SPINE Sensor interface. The driver must also self-register, possibly at boot time, to the SensorRegistry, as shown below.

```
call SensorsRegistry.registerSensor(ECG_SENSOR);
```

4.3. Performance evaluation

To test and evaluate the SPINE implementation discussed above, the Tmote Sky (telosb) platform and the TinyOS 2.0.2 operating system were selected. Some results are described in the following. In particular, the performance evaluation has involved:

- times for significant SPINE processing operations;
- memory usage of SPINE components;
- channel bandwidth usage.

To measure the actual elapsed time, the hardware counter register directly connected to the built-in 32KHz crystal oscillator was considered.

Table V reports the execution times for some TinyOS basic operations. The results can be seen as a scale for SPINE operations processing time.

Table VI shows the processing times of some feature extractors provided by SPINE. Features compute on sensor data are already present in buffers, hence the results are not affected by the sensor sampling time.

These experimental data about the time needed by different services on the node can be used during the design phase to determine the processing capabilities of the nodes and define task allocations that avoid data losses. In particular, since TinyOS is a single task operating system, a task execution will prevent other SPINE functionalities (if they are not forced to pre-empt on others) and therefore the network manager (coordinator) must carefully activate on node functionalities for avoiding data losses.

Evaluating the memory consumption of an extensible, customizable framework like SPINE requires some observations because significant memory occupancy resides outside the framework

Table V. Some TinyOS basic operations.

Operation	Time (ms)
Radio start-up	2.685
Radio shut-down	0.244
Packet transmission (one active message with 28-byte payload)	From 5.13 to 24.26 (mean 10.07)
ST LIS3LV02DQ accelerometer sampling (all 3-axis)	1.68
MSP430 voltage diode sampling	17.48

DOI: 10.1002/spe

Table VI. Processing time for some feature extractors in SPINE.

Operation	Time (ms)		
	200 samples	100 samples	50 samples
Max (on 3 sensor channels)	1.67	0.88	0.49
Mean (on 3 sensor channels)	2.68	1.65	1.1
Standard deviation (on 3 sensor channels)	53.70	16.80	16.99
Vector magnitude (on 3 sensor channels)	4.58	2.89	2.44
Pitch & roll (on 3 sensor channels)	19.53	18.37	17.90
Entropy (on 3 sensor channels)	1016.39	488.18	239.30

Table VII. Memory requirements for SPINE configurations.

SPINE Configuration	ROM (bytes) Max 48 KB	RAM (bytes) Max 10 KB
Core only	14 904	$1500+2\cdot buffer\_size\cdot(buffer\_pool\_size+1)$
Motion sensor-board + features/alarms	34 316	3860

Table VIII. Radio channel usage in different processing configurations.

Sampling rate	RawData_Rate	Window	Sliding (%)	Feature rate	Savings (%)
40 Hz	40 pkt/s	80 samples	50	1 pkt/s	97.5
40 Hz	40 pkt/s	80 samples	25	2 pkt/s	95
10 Hz	10 pkt/s	20 samples	50	1 pkt/s	90
10 Hz	10 pkt/s	20 samples	25	2 pkt/s	80

core and within its extensions (e.g. sensor drivers, processing functions); moreover, it is in part influenced by some system configuration parameters (e.g. sizing of buffers).

Table VII shows the memory usage of the SPINE 1.2 core with an extension for using a motion sensor board and computing Features (Max, Min, Range, Mean, Amplitude, Median, Mode, RMS, Variance, Standard Deviation, Total Multi-Channel Energy, Vector Magnitude, Pitch & Roll, Entropy) and Alarms (thresholds-based events on Features values).

Radio channel usage depends a lot on the application design choices. In fact, in SPINE, data sampled by sensors on the motes may be sent to the coordinator without any processing (raw data) and then be analyzed on the coordinator, or, once clear which processing is needed, part of it can be done on nodes to achieve a better solution in terms of energy consumption and channel optimization. Therefore, data may be pre-processed on the node and only the computation result sent to the coordinator.

Users may decide to collect all the data coming from the sensors and then process them at the coordinator, therefore a SPINE data packet will be sent every time the sensor is sampled.

$$RawData\_Rate = Sampling\_Rate$$

However, if on-node feature computation capabilities are used, once the most significant features are selected and activated, every node will send a data packet to every window sliding.

$$Feature\_Rate = \frac{Sampling\_Rate}{Window \cdot Sliding\%}$$

Table VIII reports a few examples showing that the saving in terms of pkt/s can be significant.

DOI: 10.1002/spe

5. THE JAVA-BASED SPINE COORDINATOR

As discussed in Section 3.2, the SPINE architecture consists of two entities, one on the sensor nodes and the other on a coordinator station. Such a coordinator can be a desktop computer, a laptop or even a PDA or a smart-phone. The coordinator provides the end-user application with an access point to the wireless BSN. Thus, its main tasks are controlling the remote nodes and capturing the various messages and events generated, according to the user-application needs.

The design of the SPINE coordinator is driven by principles of lightweight, ease of use and portability. The idea is to provide a small set of high-level operations by which the BSN can be controlled and an effective set of events to let the application be notified of information coming from the BSN. To enhance portability, the Java language was adopted. As is well known, Java is supported not only by PCs, but by most of the current PDA and smart-phones. Hence, a careful use of Java libraries and paradigms allows fast porting, e.g. from a PC implementation of the SPINE coordinator to a mobile phone. In particular, the implementation is based on the 1.4 version of Java and makes use of data structures and libraries that are available both for the desktop and mobile edition of the Java Virtual Machine.

Before discussing a simplified architecture of the coordinator-side of SPINE, it is important to observe that none of the existing computer and mobile phones have a native wireless interface for communicating with sensor nodes. In fact, most of them are based on the IEEE 802.15.4 communication protocol [26], rather than Wi-Fi or Bluetooth. Hence, a portion of the implementation is strictly dependent on the particular base-station module attached to the coordinator node to allow the physical communication with the BSN nodes.

Figure 12 shows a simplified Package Diagram of the SPINE Coordinator. The SPINE Core package includes the SPINE Manager class contained in the Commands API and used by end-user applications for issuing commands to the BSN. Moreover, the SPINE manager is responsible for capturing low-level messages and nodes events through the Event Listener to notify registered applications with higher level events and messages content. Additionally, the SPINE Core package contains tables of constant codes, such as sensors and functions codes, which must be aligned with the ones present on the nodes.

The SPINE Datamodel package contains classes that represent the high-level, platform independent SPINE Messages as well as abstractions of BSN nodes and sensor itself (e.g. a Node object will be characterized by attributes as its type and id and built-in sensors lists and available processing functions lists).

End-user applications are only aware of the SPINE Core and Datamodel packages and hence are completely decoupled by specific implementations of the SPINE Messages and communication procedures of the currently available sensor node platform.

The SPINE Communication package is internally composed of a Send/Receive interface and some components implementing that interface according to the specific base-station platform and that represent the high-level SPINE Messages in platform-specific messages. Thus, most of the work for porting an implementation of the SPINE Coordinator on a different platform is to be carried out in this package. Currently, an implementation supporting TinyOS sensor devices is fully available, while a porting for the ZigBee-compliant platform, TI Z-Stack [9], has been also developed and is currently under testing.

6. A CASE STUDY: HUMAN ACTIVITY RECOGNITION

The SPINE framework has been used to design a human activity monitoring system prototype. This application is able to recognize postures (e.g. lying, sitting or standing still) and a few movements (e.g. walking and jumping) of a person; furthermore, it can detect if the monitored person has fallen and unable to stand up.

The wearable nodes are based on the Tmote Sky platform [27] to which is attached a custom sensor board (SPINE sensor-board) including a 3-axis accelerometer and two 2-axis gyroscopes.

DOI: 10.1002/spe

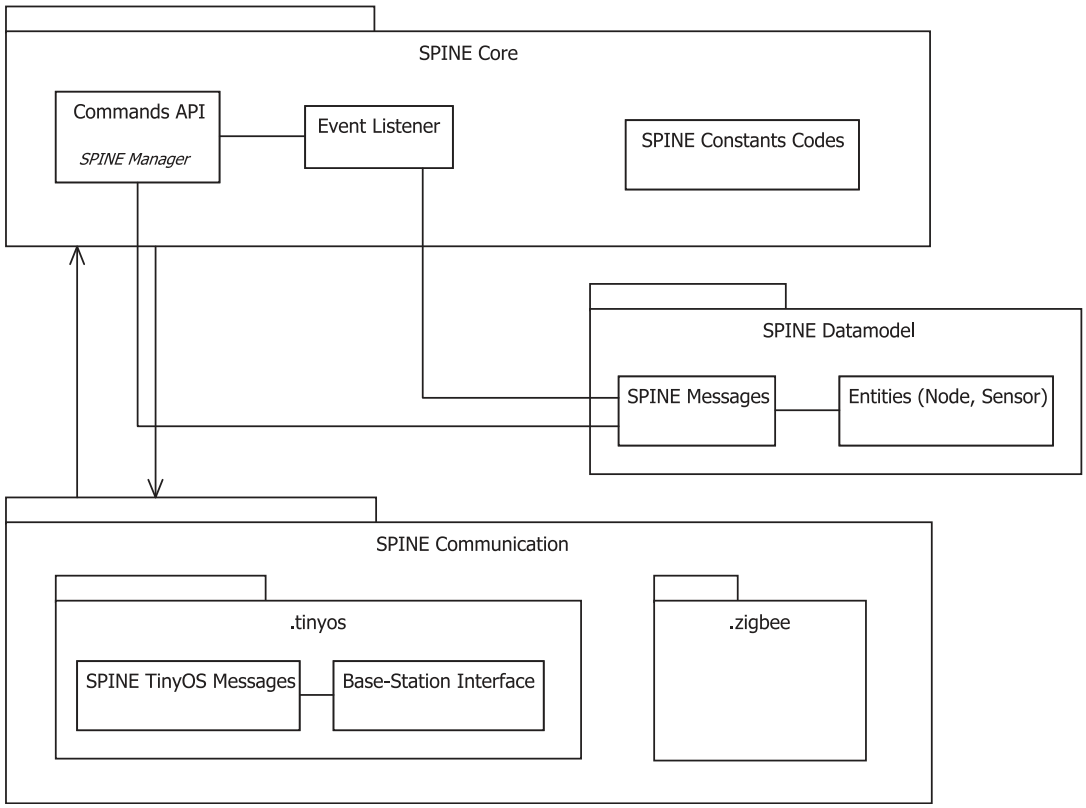


Figure 12. Simplified package diagram of the SPINE coordinator.

The nodes are powered by a standard 3.7 V, 600 mAh Li-Ion camera battery). The user application is implemented in the Java language and runs on top of a SPINE coordinator laptop to which is attached a Tmote Sky, acting as a base-station bridge, connected via USB port. The activity recognition system prototype relies on a classifier that takes accelerometer data measured by sensors placed on the waist and on a leg of a person and recognizes the movements defined in a training phase. Among the classification algorithms available in the literature, a K-Nearest Neighbor [28] (KNN)-based classifier was defined.

The prototype provides a default training set and a graphical wizard to let the user build his own training set to enhance recognition accuracy. The significant features to be activated on the node to classify the movements are then selected using an offline sequential forward floating selection [29] (SFFS) approach or the naïve sequential forward selection (SFS), embedded in the application prototype, running much faster but with worse results. The experimental results show that, given a certain training set, the classification accuracy is not much affected by the K value or the type of distance metric used by the classifier. This is because, in this specific example, classes (lying, sitting, standing and walking) are rather separate and not affected by noise. Therefore, $K = 1$ and the Manhattan distance as parameters of the KNN-based classifier were set up.

The experiments have been performed using two sensor nodes: one placed on the waist and the other on the thigh of the right leg. Then, an SFFS off-line was executed to select the smallest set of features to be activated on the nodes to achieve sufficiently accurate classification. In the feature selection algorithm the accuracy hwas calculated with a shift of 50% of the data window, taking into account half of the data set for training and half to test the classifier.

The resultant most significant features are:

- *waist node*: mean on the accelerometer axes XYZ, min value and max value on the accelerometer axis X;
- *leg node*: min value on the accelerometer axis X.

DOI: 10.1002/spe

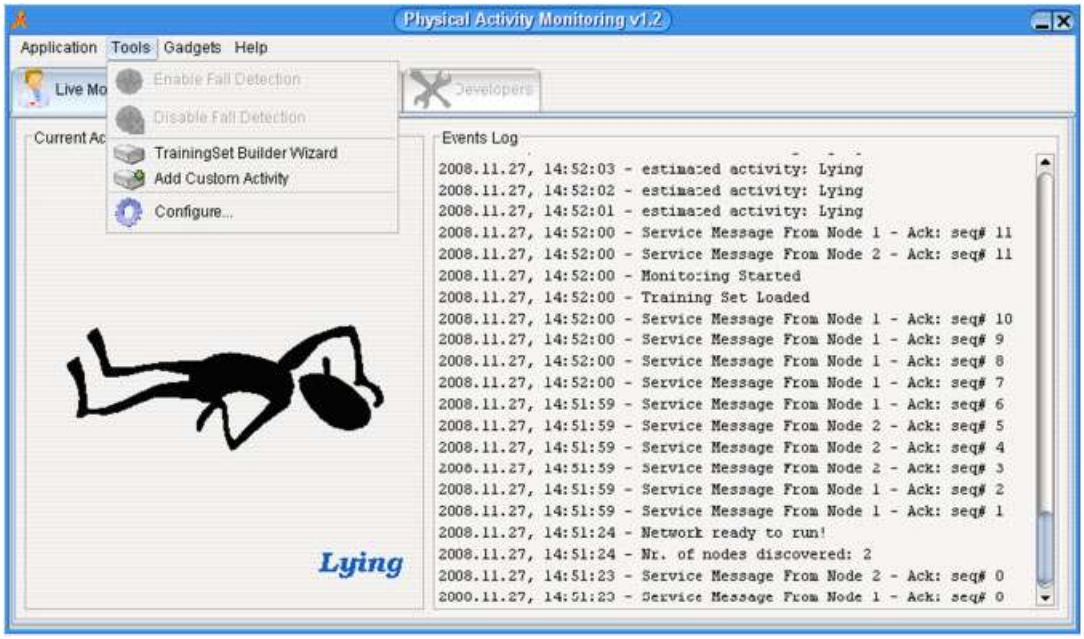


Figure 13. Live monitor panel.

The fall detection is implemented on the waist sensor node and can be activated/deactivated at run-time. When the fall detector is active, the Alarm Engine (see Section 4.1), every time new accelerometer data are acquired, checks whether the cross-axial energy feature, calculated on the three accelerometer axes, is greater than an empirically evaluated threshold and, in the positive case, sends an alarm message back to the coordinator to inform the user application. False alarms are drastically reduced by a simple mechanism implemented within the user application: as soon as it receives a fall-detected message, it waits the recognition of the next seven postures of the person; only if it evaluates 4 out of 7 lying positions is an emergency message reported to the user attention.

An interesting functionality of the prototype is a simple tool for adding new, user-defined activities among the default ones. The tool drives the user through a simple procedure for acquiring the necessary training data which are then stored in the global data set. Another tool of the application is a graphical step-counter with daily progress indicator which relies on a simple on-node step recognition algorithm that runs on the waist node.

The user application is composed of the following graphical panels:

- *Live Monitor* panel (see Figure 13), which allows to monitor the activity of the subject in real-time by also visualizing the timed log of the main events occurred.
- *Statistics* panel, (see Figure 14) which shows the percentages of time of the different kinds of activities the subject is performing.
- *Advanced* panel (see Figure 15), which shows, for each sensor node currently exploited, the sensor types, the available and the enabled feature extractors, and the battery voltage level.
- *Developers* panel (see Figure 16), which is intended for debugging purposes as well as for node functionality test. The panel includes a graph to plot either raw data sensor readings or feature values; a textual log for alarms and other node system messages is also included.

Although the objective of this prototype concerned mainly in testing the SPINE framework in a semi-realistic use case, the overall performance (see Table IX) reached by the recognition system is considerably high, with an average posture/movement classification accuracy of 97%. The fall detection algorithm is quite accurate as well with almost zero undetected falls and a low percentage of false alarms.

DOI: 10.1002/spe

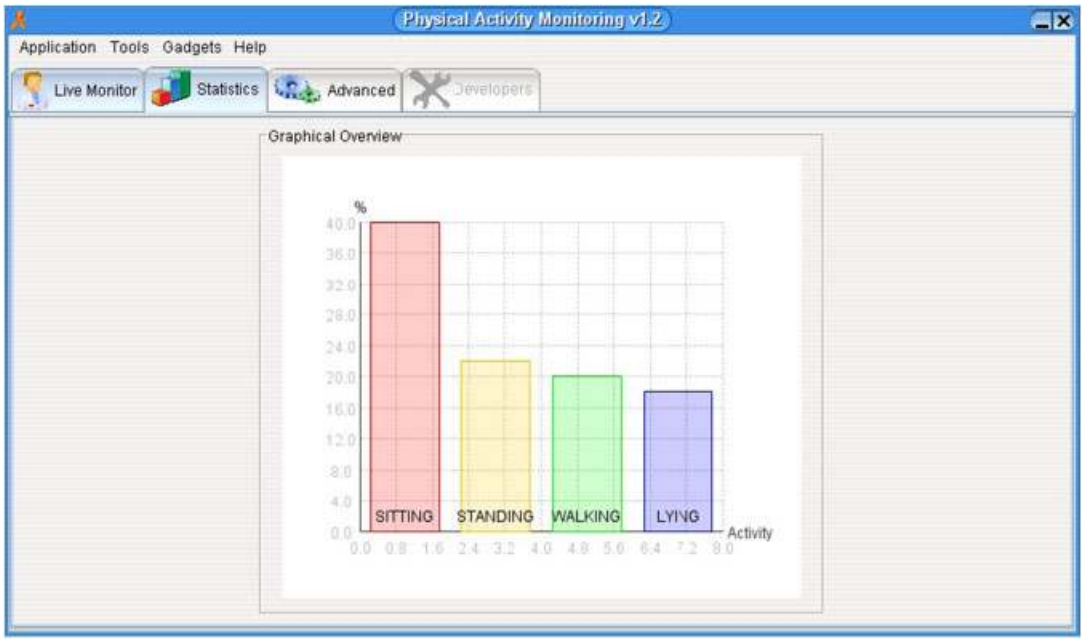


Figure 14. Statistics panel.

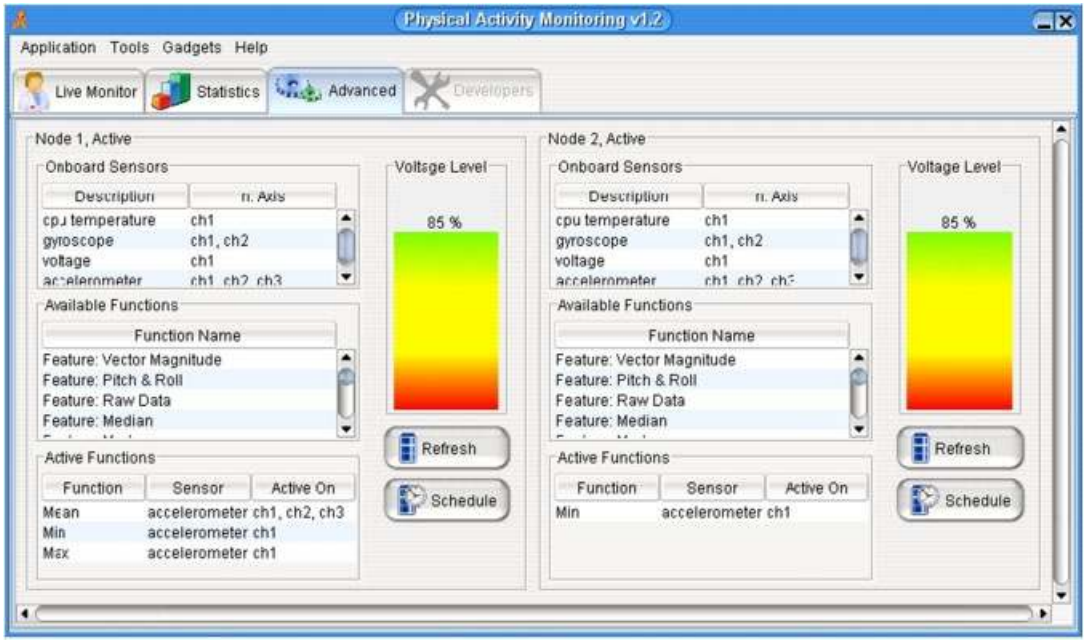


Figure 15. Advanced panel.

6.1. An analysis of the system development effectiveness and performance

This section is devoted to provide an analysis of the development effectiveness and performance of the proposed human activity monitoring system. In particular, the analysis is carried out with respect to an application developed without any high-level framework and centered on an approach based on the processing of sensed data at the base station side only.

Given the chosen parameters (sampling time = 50 ms, window = 40 samples, shift = 20 samples), the communication bandwidth is 20 pkt/sec (in bit/s) if only raw data readings are

DOI: 10.1002/spe

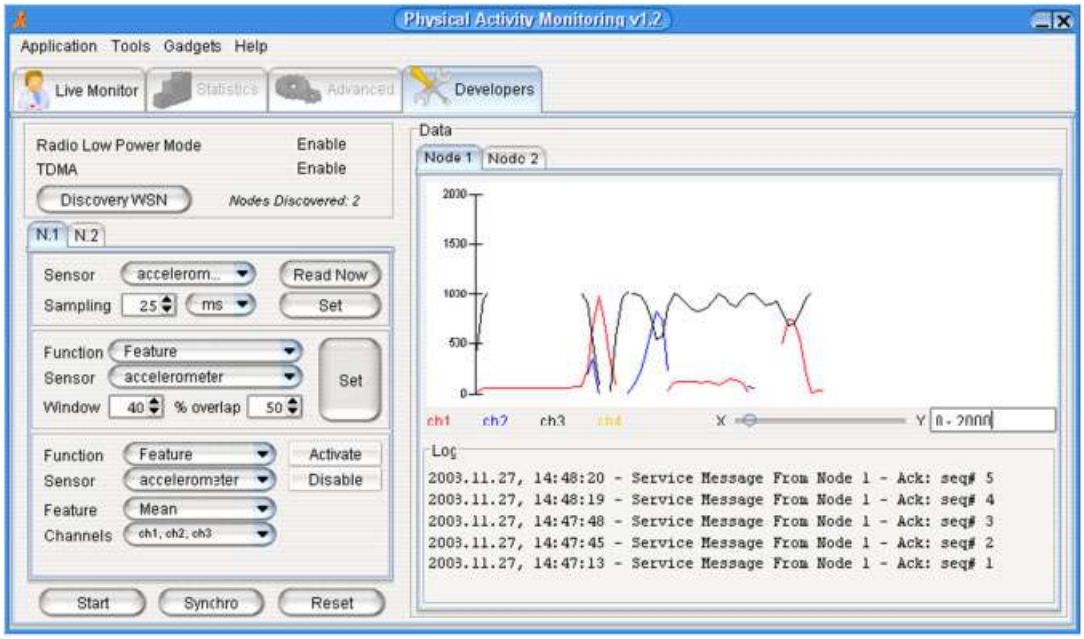


Figure 16. Developers panel.

Table IX. Posture/Movement recognition accuracy.

Sitting	Standing	Lying	Walking	Falling
96%	92%	98%	94%	100%

transmitted, while the bandwidth drops to just 1 pkt/s in the implementation which uses the in-node feature extraction as new features are extracted each *shift* sampling time* = 1 s. Therefore, by enabling the in-node feature computation, an overall bandwidth saving of 95% is achieved. On the basis of the same operating parameters, the experimental results showed that the battery lifetime is less than 24 hours of continuous monitoring when raw data readings are transmitted, while it is almost 6 days if only the most significant features are computed on-node and then transmitted. Thus, the radio duty cycling along with its in-node signal processing capabilities provided an improvement of almost 6 times that of the battery lifetime. This result does not reflect exactly the bandwidth saving mainly because the implemented duty cycling mechanism keeps the radio on listening for incoming packets for a certain period; furthermore, the computational load (i.e. the microcontroller usage) increases when extracting the features on the nodes.

In order to compare the development effectiveness of the SPINE system with the non-SPINE-based system, the basic components of a BSN application and the specific ones related to the developed human activity monitoring system were analyzed and the critical development points identified by defining the percentage of efforts for developing each component node-side and base-station side with respect to the complete application node-side and base-station side, respectively (see Figure 17). As can be seen, without SPINE, all the components had to be developed, whereas using SPINE, only some components had to be developed since many were already available in the libraries and easily configurable at compile-time and run-time. The saved development efforts are 100% at the node side, if only the default components are used, and 80% at the base-station side, in case of only the classification algorithm being implemented; this consideration shows that a notable improvement can be obtained by adopting SPINE for the development of BSN applications.

DOI: 10.1002/spe

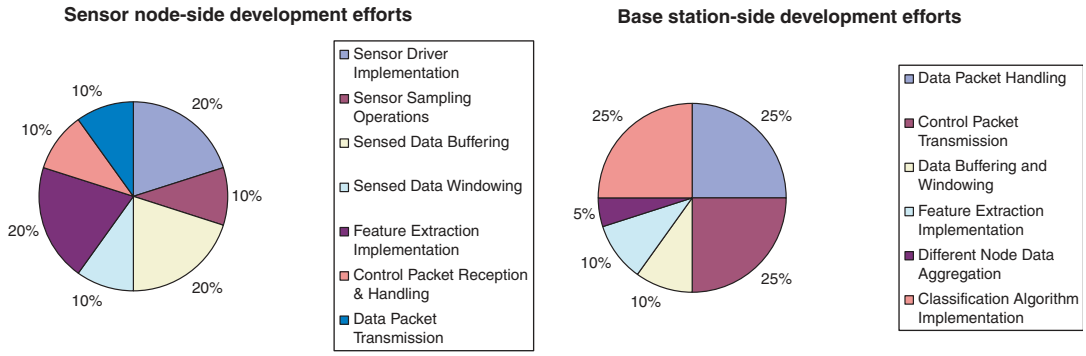


Figure 17. Development efforts for typical BSN applications at sensor node-side and base station-side.

7. CONCLUSIONS

The development of BSNs applications is a complex task which requires suitable frameworks and tools for effective and efficient programming at the base station and sensor node sides. In this paper SPINE, a domain-specific framework for the rapid prototyping of signal-processing-oriented BSN applications has been presented. SPINE is split into two parts: the base station side and the sensor node side. The base station is based on a flexible Java framework which allows for interaction with (configuring, starting, monitoring and data exchanging) the sensor nodes and performing data monitoring and visualization, and complex data classification and analysis. The sensor node is based on an efficient framework, currently implemented in nesC on TinyOS, which provides sensing, processing and communication operations.

The performance evaluation of SPINE at the sensor node side confirms that it is efficient enough to support intensive sensing and data processing applications. Currently SPINE is applied in the health care domain for human activity monitoring. In particular a real-time system for the recognition of posture and movements (standing, sitting, lying, walking, falling) has been successfully developed and described. Moreover, it has been shown an effectiveness and performance comparison between the SPINE-based application and the same application developed without signal in-node processing. SPINE not only decreases the development efforts but also allows to obtain higher performances with respect to energy consumption and used network bandwidth.

The experience being gained in the context of the SPINE project for the development of SPINE and its applications in the health care domain, has already highlighted the need for software abstractions and platforms to support an effective development of BSN-based systems. The capability to perform real-time processing of the sensed data directly on the sensor node should be considered a requirement for BSN frameworks/applications as well as flexible mechanisms, node-side and base-station-side, for extension and customization to meet specific application requirements. In particular, BSN applications are based on wearable sensors, particularly of the motion (accelerometers) and biomedical (ECG, heart rate) type, which are sampled very frequently as they have to monitor fast variations of the followed phenomenon such as an irregular heart rate or a human fall; as a consequence, conspicuous data streams are produced. Thus, sending raw data from the sensor nodes to the coordinator is not efficient in terms of battery lifetime and usage of the wireless medium. In fact, the radio is one of the most energy consuming components of a sensor node and consumes significantly not only when transmits or receives data but also when it listens to the channel waiting for incoming packets. Hence, transmissions of only significant data and/or aggregated data, and duty cycling techniques are fundamental to extending the battery lifetime. Furthermore, sending raw sampled data will likely cause packet collisions on the wireless medium. The distributed in-node processing approach of SPINE perfectly meets these requirements. Moreover, programming the application logic of the BSN system (base-station and sensor-node side) through a Java API would allow developers to concentrate on the problem solving algorithms and avoid using low-level programming languages for programming the sensor nodes. The SPINE

DOI: 10.1002/spe

Java API allows to remotely configure and control all the in-node SPINE services, thus supporting a more rapid and robust software application development. This is one of the main strengths of SPINE: algorithm and application developers do not have to deal with embedded programming but can manage and configure the processing functionalities of the sensor nodes using a simple and more familiar Java interface. Additionally, the flexible architecture of the SPINE-based sensor node software allows interested and skilled developers to more easily program in-node functionalities as well as to more rapidly introduce new sensors and hardware platforms.

Owing to the aforementioned advantages several research groups have already adopted SPINE for the development of novel BSN applications and systems. In [30, 31] SPINE has been used for prototyping an online gait analysis system based on Hidden Markov Model using wireless sensors placed on the legs and in the shoes. In [32] the porting of SPINE on a Nokia N810 PDA has been carried out for collecting accelerometer data of multiple sensors placed on subjects walking around the UCB campus. This effort is part of a larger system, named DexterNet, which also integrates GPS data and pollution measurement data. In the department of biomedical engineering at the Tampere University of Technology, SPINE has been used to develop a firmware, which is now integrated into the SPINE framework, for a bio-sensor board attachable to TelosB nodes and mounting an ECG sensor, an Electrical Impedance Pneumography (EIP) sensor and a three-axis accelerometer sensor [32]. In the context of the 'OpenCare Project' [33] SPINE is used to easily integrate specific hardware mote platforms as well as to program applications based on signal in-node processing. Finally, at the Computer and Embedded System Laboratory of the National School of Engineers of Sfax in Tunis, SPINE is being extended for supporting nodes authentication.

On the basis of the results described in this paper, on-going work is aimed at finalizing the porting of SPINE on Texas Instruments Z-Sstack [9], a ZigBee-compliant sensor platform, and testing SPINE on the Shimmer nodes [24], another TinyOS-based sensor platform. As the SPINE-based BSN application development targets a specific platform, SPINE2 has been designed for development of BSN applications on heterogeneous sensor platforms and is going to be implemented and tested on TinyOS, Z-Stack and Ember [34] sensor platforms. Finally cooperative mechanisms among BSNs based on mutual interaction between their SPINE coordinators and their sensor nodes are being investigated to enable hand-off of assisted livings among multiple coordinators and non-supervised information exchange.

ACKNOWLEDGEMENTS

The authors thank Roozbeh Jafari at the University of Dallas, Sameer Iyengar, Kevin Klues, and Alberto Sangiovanni-Vincentelli at the University of Berkeley, Filippo Tempia Bonda at Telecom Italia, Trevor Pering at Intel Research Santa Clara, Philip Kuryloski at the Cornell University, Luigi Buondonno, Antonio Giordano, Stefano Galzarano at the University of Calabria, for their precious contributions to the SPINE project in terms of ideas, discussions and implementation efforts. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

REFERENCES

1. Akyildiz IF, Su W, Sankarasubramaniam Y, Cayirci E. Wireless sensor networks: A survey. *Computer Networks*: *The International Journal of Computer and Telecommunications Networking* 2002; **38**(4):393–422.
2. Yang G-Z. *Body Sensor Networks*. Springer: New York, 2006.
3. Fok C-L, Roman G-C, Lu C. Mobile agent middleware for sensor networks: An application case study, In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks* (*IPSN'05*), Los Angeles, CA, 25–27 April 2005; 1–6.
4. Kumar R, Wolenetz M, Agarwalla B, JunShin S, Hutto P, Paul A, Ramachandran U. DFuse: A framework for distributed data fusion. *ACM SenSys*, Los Angeles, CA, U.S.A., 5–7 November 2003; 114–125.
5. Wendi Heinzelman B, Amy Murphy L, Carvalho HS, Mark Perillo A. Middleware to support sensor network applications. *IEEE Network* 2004; **18**:6–14.
6. Madden S, Michael Franklin J, Hellerstein J, Hong W. TAG: A tiny aggregation service for *ad-hoc* sensor networks. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (*OSDI '02*), Boston, MA, U.S.A., 9–11 December 2002; 131–146.

DOI: 10.1002/spe

7. Souto E, Guimarães G, Vasconcelos G, Vieira M, Rosa N, Ferraz C, Kelner J. Mires: A publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing* 2006; **10**(1):37–44.
8. SPINE website. Available at: http://spine.tilab.com, 2009.
9. Z-Stack website. Available at: http://focus.ti.com/docs/toolsw/folders/print/z-stack.html. 2009.
10. Simon D, Cifuentes C. The squawk virtual machine: Java^{TM} on the bare metal. *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (San Diego, CA, U.S.A., October 16–20, 2005), *OOPSLA '05*. ACM: New York, NY, 2005; 150–151.
11. Malan D, Fulford-Jones T, Welsh M, Moulton S. CodeBlue: An *ad hoc* sensor network infrastructure for emergency medical care. *MobiSys 2004 Workshop on Applications of Mobile Embedded Systems* (*WAMES 2004*), Boston, MA, 6 June 2004; 12–14.
12. Lombriser C, Roggen D, Stager M, Troster G. Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. *Verteilten Systemen* (*KiVS 2007*), Bern, Switzerland, 26 February–2 March 2007; 49–57.
13. TinyOS website. Available at: www.tinyos.net. 2009.
14. Najafi B, Aminian K, Ionescu A, Loew F, Büla CJ, Robert P. Ambulatory system for human motion analysis using a kinematic sensor: Monitoring of daily physical activity in the elderly. *IEEE Transactions on Biomedical Engineering* 2003; **50**(6):711–723.
15. Hester T, Hughes R, Sherrill DM, Knorr B, Akay M, Stein J, Bonato P. Using wearable sensors to measure motor abilities following stroke. *Proceedings of the 3rd International Workshop on Wearable and Implantable Body Sensor Networks*, (*BSN 2006*), MIT, Boston, MA, U.S.A., 3–5 April 2006; 5–8.
16. Pansiot J, Stoyanov D, McIlwraith D, Benny Lo PL, Yang GZ. Ambient and wearable sensor fusion for activity recognition in healthcare monitoring systems. *Proceedings of 4th International Workshop on Wearable and Implantable Body Sensor Networks* (*BSN 2007*), RWTH Aachen University, Germany, 26–28 March 2007; 208–212.
17. Maurer U, Smailagic A, Siewiorek DP, Deisher M. Activity recognition and monitoring using multiple sensors on different body positions. *Proceedings of the 3rd International Workshop on Wearable and Implantable Body Sensor Networks* (*BSN 2006*), MIT, Boston, MA, U.S.A., 2006; 113–116.
18. Lester J, Choudhury T, Borriello G. A practical approach to recognizing physical activities. *International Conference on Pervasive Computing* (*PERVASIVE*), Dublin, Ireland, 2006; 1–16.
19. Bao L, Stephen Intille S. Activity recognition from user-annotated acceleration data. *Proceedings of the 2nd International Conference on Pervasive Computing* (*PERVASIVE*), Vienna, Austria, 21–23 April 2004; 1–17.
20. Sadilek DA. Prototyping domain-specific languages for wireless sensor networks. *Proceedings of the 4th International Workshop on Software Language Engineering*, Nashville, TN, U.S.A., 2007; 237–241.
21. LGPL documentation. Available at: http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html, 2009.
22. Giannantonio R, Gravina R, Kuryloski P, Seppä V-P, Bellifemine F, Hyttinen J, Sgroi M. Performance analysis of health care systems based on the spine framework. *Proceedings of the 3rd International Conference on Pervasive Computing Technologies for Healthcare* (*Pervasive Health'09*), London, U.K., 1–3 April 2009.
23. MicaZ website. Available at: http://www.xbow.com/Products/productdetails.aspx?sid=164, 2009; 1–8.
24. Shimmer website. Available at: http://www.shimmer-research.com/, 2009.
25. Bramer M. *Principles of Data Mining*. Springer: London, 2007.
26. 802.15.4 Website. Available at: http://www.ieee802.org/15/pub/TG4.html, 2009.
27. Moteiv website. Available at: http://www.sentilla.com/moteiv-endoflife.html, 2009.
28. Cover T, Hart P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 1967; **13**:21–27.
29. Pudil P, Novovicova J, Kittler J. Floating search methods in feature selection. *Pattern Recognition Letters* 1994; **15**(11):1119–1125.
30. Raveendranathan N, Loseu V, Guenterberg E, Giannantonio R, Gravina R, Sgroi M, Jafari R. Implementation of virtual sensors in body sensor networks with the SPINE framework. *IEEE Symposium on Industrial Embedded Systems* (*SIES 2009*), Lausanne, Switzerland, 8–10 July 2009; 124–127.
31. SPINE HMM. Available at: http://www.essp.utdallas.edu/Main/OpenSource#spine, 2010.
32. Kuryloski P, Giani A, Giannantonio R, Gilani K, Gravina R, Seppä V-P, Seto E, Shia V, Wang C, Yan P, Yang AY, Hyttinen J, Sastry S, Wicker S, Bajcsy R. DexterNet: An open platform for heterogeneous body sensor networks and its applications, Body Sensor Networks (BSN 2009), Berkeley, CA, U.S.A., 3–5 June 2009; 92–97.
33. OpenCare Project—SPINE. Available at: http://opencareproject.wikispaces.com/SPINE, 2010.
34. Ember website. Available at: http://www.ember.com, 2009.

DOI: 10.1002/spe