Performance Analysis of an Activity Monitoring System using the SPINE Framework

Roberta Giannantonio[§], Raffaele Gravina*, Philip Kuryloski*, Ville-Pekka Seppä [†], Fabio Bellifemine[§], Jari Hyttinen[†], Marco Sgroi*
[§]Telecom Italia, Torino, Italy
*WSN Lab sponsored by Pirelli and Telecom Italia Berkeley, CA
[†]Department of Biomedical Engineering, Tampere University of Technology, Finland

Abstract — SPINE is an Open Source Framework for the design of signal processing intensive (SPI) WSNs. It supports the construction of WSN applications through high-level abstractions and libraries, and allows designers to quickly explore implementation tradeoffs through fast prototyping. This paper describes the architecture of SPINE and presents implementation parameters, such as processing time, memory, bandwidth usage and power consumption that are most relevant for application developers to set tunable parameters and analyze system performance. Finally, the paper presents the performance and the resource usage of a SPINE based posture recognition system for elderly health monitoring.

Keywords-component; wireless sensor networks, performance analysis, SPINE

I. Introduction

The design of health care systems based on Wireless Sensor Networks (WSNs) is complex, particularly due to the challenge of implementing signal processing intensive algorithms for data interpretation on wireless nodes that are very resource limited and have to meet hard requirements in terms of wearability and battery duration. New abstractions and tools are needed to support application developers during design space exploration and fast prototyping.

A small number of design frameworks have been proposed to address this problem. Titan [1] supports context recognition in dynamic sensor networks. Context recognition algorithms are represented as a network of interconnected data processing tasks. Titan adapts to different context recognition algorithms by dynamically reconfiguring individual sensor nodes to update the network wide algorithm execution. CodeBlue [2] is a framework built on TinyOS designed to provide routing, naming, discovery, and security for wireless medical sensors, PDAs, PCs, and other devices that may be used to monitor and treat patients in a range of medical settings. The same team has developed a new operating system for data-intensive sensor network applications called Pixie [3] that enables resource-aware programming and allows applications to receive feedback on resource usage, and control resources. The Pixie OS gives visibility and fine-grained control over resource management through the concepts of resource tickets, a core abstraction for representing resource availability and reservations, and resource brokers, which mediate between low-level physical resources and higher-level application demands.

SPINE (Signal Processing in Node Environment) [4] is a Framework for the design of signal processing intensive (SPI) WSNs. It provides higher abstraction levels and support to quickly explore implementation tradeoffs through fast prototyping. Titan and Pixie focus on allocation and management of resources, where SPINE focus on making a variety of functionality easily and conveniently accessible, yet configurable by designers. SPINE is based on the following principles:

- Open Source. The SPINE project [5] is Open Source to establish a community of users and developers. The SPINE code is available under the LGPL license.
- Interoperability through APIs. SPINE provides local and remote applications with lightweight Java APIs that they can use to manage the 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 libraries of protocols, utilities and data processing functions and 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 TinyOS.
- Distributed implementations of data processing and interpretation algorithms. SPINE helps designers to evaluate the efficiency of distributed implementations of data classification algorithms with respect to the use of energy and channel bandwidth.

This paper provides a detailed description of the architecture of the last release of SPINE (1.2 version) and focuses on the performance and cost analysis aspects of the design of an application using SPINE. In particular, it provides description of parameters, such as processing time, memory, bandwidth usage and power consumption that are most relevant for application developers to set tunable parameters properly and analyze system performance. Finally, it shows how these parameters are used in the performance analysis of a posture recognition system for elderly health monitoring.

Digital Object Identifier: 10.4108/ICST.PERVASIVEHEALTH2009.6004
http://dx.doi.org/10.4108/ICST.PERVASIVEHEALTH2009.6004

II. Architecture of SPINE

A. Network Architecture

The network architecture of a WSN supported by SPINE includes one Coordinator Node (CN) and one or more Sensor Nodes (SN). The CN manages the network, collects and analyzes the data received from the SNs, and acts as a gateway to connect the WSN with wide area networks for remote data access. SNs measure local physical parameters and send raw or interpreted data to the CN. Fig. 1 visualizes the architecture of a SPINE network.

Currently SPINE supports WSNs with star topology, where each SN communicates only with the CN. However, the Framework can be easily extended to support also multi-hop communication and direct communication among SNs. In the current version of SPINE a SN can be associated with a single gateway. However, a possible extension of the Framework is to allow nodes to leave a WSN and join another WSN managed by a different CN. This scenario might occur when a patient wearing body sensors moves across locations and at each location connects to a different gateway.

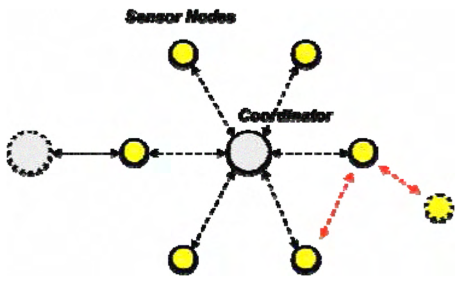

Fig. 1. SPINE Network Architecture

B. Functional Architecture

The functional architecture of the SPINE Framework (Fig. 2) consists of two main components: the Service Layer and the Communication Layer.

The Service Layer of SPINE includes the following components: the Service Layer Manager, the Sensor Data Collection Service and the Sensor Data Interpretation Service.

The Service Layer Manager implements the main logic of the Service Layer and dispatches computation among services.

The Sensor Data Collection Service manages sensor data sampling and buffering on the sensor nodes. In particular, the Sensor Controller manages:

- real-time activation/deactivation of sensors or independent channels.
- the sampling rate, which can be set dynamically (e.g. for more detailed monitoring when a relevant event has been detected).

The Sensor Controller wraps each supported sensor with a common interface. As a result, each sensor appears as variation of a similar abstract component to processing and communication components of SPINE. This allows new sensors to be introduced without requiring changes to other SPINE components. The Buffer Manager manages the buffers that are allocated at design time, one for each active signal acquisition channel. Buffers are shared among the active functions on a node so that multiple functions can access the same stored data during any processing interval.

The Sensor Data Interpretation Service includes functions that are used to process or interpret the raw sensor data provided by the Sensor Data Collection Service. This service includes:

- libraries of features implemented on the sensor nodes such as mean, median, central value, amplitude, range, minimum, max, root mean square, standard deviation, variance and cross-axial energy. Standard math functions are also implemented on the sensor nodes to allow the designer to define additional feature extractors as necessary. A Feature can operate over all active channels of a given sensor and can produce an array of data to be sent to the base station
- a Feature Extraction Engine which allows activation or deactivation of the computation of features for a given signal acquisition channel at runtime. Furthermore, it allows dynamic tuning of the following two feature extraction parameters: window size (time interval on which the feature is to be calculated), and window shift (the period at which each feature has to be computed).
- an Alarm Engine which generates events when certain conditions determined by the comparison of local variables with predefined thresholds occur. Alarms can be set on any of the supported features, including raw data, specifying window and shift settings as well as alarm-specific parameters.

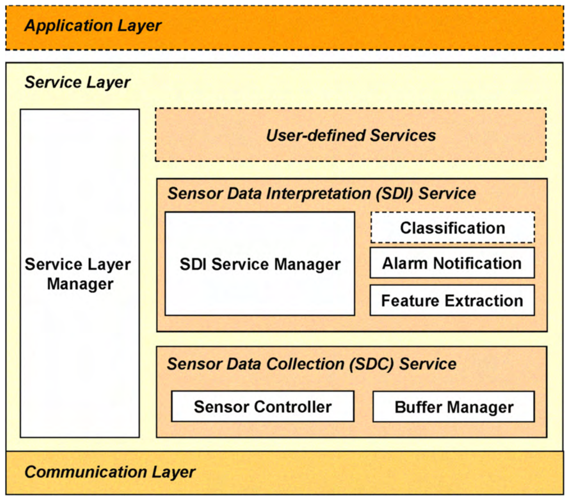

Fig. 2. SPINE Functional Architecture

Other feature extraction or classification functions in addition to those currently included in SPINE may be required for specific applications. The Framework makes it easy to add User-defined Services.

The Communication Layer manages the communication between the coordinator and the sensor nodes. It includes:

- a Radio Controller that manages the duty cycle of the radio and switches the radio on when data is ready to be transmitted;
- an end-to-end protocol (called SPINE protocol) between the WSN coordinator and the sensor nodes. The protocol is used by the coordinator to activate functions on the sensor nodes and specify settings such as time intervals and thresholds and by the sensor nodes to send sensor data or the result of the functions computed on the node to the coordinator;
- a TDMA Protocol that is optional and can be enabled depending on the requirements of the target application.

III. HW/SW Platforms

The functional architecture described in the previous section is partly implemented on the SNs and partly on the CN.

The SN component is currently implemented in nesC and executed in the TinyOS [6] environment. SPINE has been structured to be platform-independent and may run on different TinyOS 2.x hardware platforms (MicaZ, Telos[7], Shimmer [8]). Fig. 3 visualizes the components of a SN.

The CN component consists of a Java-based interface that an application running on the CN itself or on a remote server can use to manage the sensor nodes or make service requests. SPINE provides a lightweight Java API that is easily portable to devices of various capabilities that can be used as gateway, such as a PC or a mobile phone.

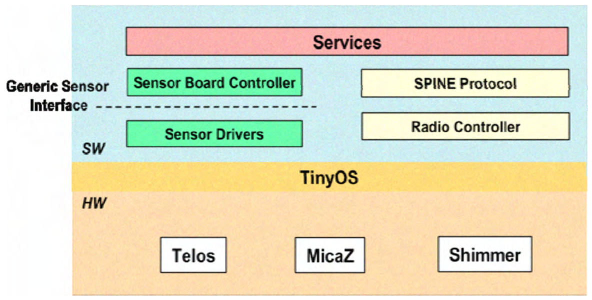

Fig. 3. Sensor Node Architecture

A. Motion Board

SPINE supports a motion sensor node (Fig. 4a). This motion node is composed of a commercially available TelosB mote on which has been connected a custom-made motion sensorboard.

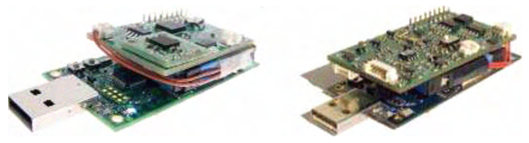

Fig. 4. a) Motion board, b) Physiological sensor board

The TelosB platform has a 8MHz microcontroller with 10K RAM and 48K ROM, 1Mb of external flash memory, a 802.15.4 radio module and a 10-pin expansion interface. The motion sensorboard consists of a triaxial accelerometer (ST LIS3LV02DQ), a biaxial gyroscope (IDG300), a power on/off switch and a LED. The node is powered by a rechargeable 600mA/h Li-Ion battery that allows approximately 20 hours of continuous sampling and raw data radio transmission. Lifetime can improve significantly by enabling the on-node processing of data and reducing wireless communication.

As the sensors introduce some measurement errors, a preliminary calibration could be necessary in certain applications, particularly using several sensor nodes.

B. Biosensor Board

For many BSN applications knowledge of the physiological state of the person is essential. This includes for instance diagnostics, rehabilitation, physiological and psychological research, and sports. With appropriate sensors BSNs can provide means for acquiring information that has not been available before, due to their ability to operate for extended periods of time during normal activities of the person. There have been multiple BSN projects that measure heart activity (ECG signal) and some that also incorporate breathing measurement. Unfortunately the projects so far have only measured the rate of breathing, neglecting a more essential parameter of breathing, namely the volume. For instance, in metabolic rate estimation, the respiration rate is a very poor indicator compared to breathing minute volume. To address this issue, we are developing a combined sensor capable of measuring not only heart and respiration rates, but also the volumetric parameters of breathing.

The developed sensor board prototype uses four electrodes connected to the surface of the ribcage. The same electrodes are used for both heart and breathing measurement. Breathing is measured through electrical impedance pneumography (EIP) . EIP involves feeding a very small constant high frequency current through the thorax and measuring the voltage created by the current. Changes in the voltage are proportional to changes in the conductivity of the thorax caused by breathing. The raw physiological signals are interpreted to derive ECG for heart rate and heart rate variability (HRV) and EIP signal for respiration rate, inspiration-expiration times, tidal volume, minute volume and their associated variabilities. With multivariate analysis these parameters can be used for higher level long-term derivations of, for example, energy expenditure. In a small preliminary test series during a 20 minutes gradually increasing treadmill exercise the system was capable of estimating respiration minute volume (ventilation) with a mean relative error between 5 and 13 % [9].

The biosensor board is shown in Fig 4b. Lower board is the TelosB mote and the upper is the analog sensing board. The battery is in between. The nearest white connector is for the four electrode cables. The custom analog measurement board is connected to a TelosB mote board which performs analog-digital conversion, signal processing and communication over the radio. To calculate all the parameters described above, sampling rates of 250 Hz and 20 Hz are needed for ECG and EIP, respectively. The system is powered by a small lithium-ion battery that lasts for approximately 20 hours in continuous measurement and raw data transmission. In general, the structure of the system is very similar to the motion board described above.

The software of the biosensor system was originally developed without using SPINE. The mote side was running on TinyOS sending custom data packets to a PC Matlab application that then parsed the data for displaying and recording. Every time something needed to be changed in the data packet format, extensive changes in code were required. Also, one had to be careful with scheduling and memory issues on the node side software while sampling, processing and sending data. This custom suite has been replaced by a SPINE 1.2 implementation. The primary effort in moving to a SPINE based solution was the creation of SPINE drivers for the ECG and EIP sensors. Additionally, SPINE server provided an easy and flexible java API to configure and use the sensorboard. Currently we have implemented only raw data access, but the interface for physiological event detection and transmission already exists. The next step is to implement the algorithms that interpret the ECG and EIP signals on-board, reducing network traffic significantly.

IV. Use of the Spine Framework

Efficient implementation of WSN applications requires appropriate allocation of the limited resources on the nodes in terms of power, memory and processing. This is especially critical in signal processing systems, which usually have large amounts of data to process and transmit. SPINE provides a flexible framework to support developers in evaluating the implementation tradeoffs along the memory, computation and energy dimensions. In particular, it supports both centralized and distributed implementations and offers developers tools to evaluate and select the architecture that is most suitable for the target application.

Centralized architectures maximize the amount of functions executed on the coordinator, which is usually implemented on a gateway or a mobile phone and therefore has more resources than sensor nodes. In a fully centralized approach, sensor nodes typically transmit the raw sensor data to the coordinator, which extracts features and executes classification algorithms. This implementation has the disadvantage of using large amounts of power and channel bandwidth for data transmission.

In distributed architectures sensor nodes share the burden of performing some classification functions. For example, sensor nodes may compute features locally and transmit them to the coordinator. Sending only the result of a computation instead of transmitting the raw sensor data might significantly reduce the amount of transmitted data and therefore allow a more efficient utilization of the wireless bandwidth and relevant savings of the energy of the nodes. In some cases further resource optimization could be achieved by implementing some classification functions on the node.

Typically, distributed implementations require more effort on the part of the developer. However, SPINE reduces this effort significantly by providing readily available features on the node. WSN applications must be designed as dynamic systems that adapt to varying interactions with the environment and can be easily extended with new features during the lifetime of the system.

Data processing requirements may vary significantly during system operation, for example when the person being monitored performs different activities. In this case the middleware should dynamically adapt and quickly configure internal services and communication accordingly. The Service Layer of SPINE supports dynamic adaptation of the data collection and interpretation services by allowing the coordinator node to send messages to the sensor nodes specifying parameters such as the sampling rate and the features that sensor nodes must compute.

WSN application developers can use the built-in features of the framework such as radio controller, the feature library and engine, or can easily add new features and services taking advantage of the SPINE APIs and management functions. For example, support for new sensors can be easily added by writing the sensor drivers so that they comply with the generic sensor interface and without having to modify the sensor controller, the buffer manager or the feature engine. This also results in the sensor being made immediately available for further processing by other parts of the SPINE framework.

The implementation of the SN component of SPINE has been designed to decouple sensing (SDC Service), data processing (SDI Service) and communication functionality. As a result, it is relatively easy to add new functionality and new sensor drivers. New features can be easily introduced as far as they conform to a defined feature interface. It is also possible to introduce new services or processing functions. Both the Feature Engine and the Alarm Engine conform to a defined function interface, which specifies that a processing component will be setup with one message from the base station, after which sub-functions may be each activated with an additional message. The format of setup messages and messages sent back to the base station is not defined by the interface, which gives the developer full freedom when developing new processing components. Functions and sub-functions are each addressed by a unique 8-bit value in the setup message, which allows for ample expansion of the SPINE Framework by the open source community.

V. System Design Parameters

Designing a WSN application requires allocation of tasks among the nodes and setting of the tunable parameters of the architecture in a way that maximizes the lifetime of the system (i.e. battery duration) and satisfies the data accuracy requirements.

Parameters that are specific to the implementation of the SPINE Framework on a selected HW/SW platform include:

- the processing time of the features and of the SPINE functions on the selected HW platform
- the required memory to store the code and the data
- the overhead of the SPINE protocol in terms of channel bandwidth
- the energy consumed to perform the tasks

Parameters that are directly tunable by the designer include:

- Sensor data sampling rate
- Window size and shift of features
- Features to compute
- Buffer allocation per sensor channel
- Adoption of TDMA access control and slots allocated for each node

Some tunable parameters can be set during system operation, while others must be set at design time. While the sampling rate, the window size and shift and the features to compute can be selected at run time based on the behavior of the system and its interaction with the environment, in the current implementation of SPINE buffers and TDMA policy must be set at design time and cannot be modified dynamically.

Typical performance evaluation concerns design choices and parameter settings such as

- if data is lost due to insufficient channel bandwidth or due to buffer overflow (in the case a node is not able to process incoming data fast enough);
- whether a contention-based or TDMA-based access protocol is best suited to the target application;
- how long the battery will last running the allocated tasks.

During performance analysis designers are concerned both with parameters that correspond to requirements of the system such as data accuracy, lifetime or latency, and with parameters related to the implementation of the system for the given test scenarios, such as buffer occupation, channel bandwidth usage and packet losses. In case an implementation of the system does not meet the requirements, the second types of parameters are important to identify bottlenecks and the required changes.

A. Processing Time

Processing time of the SPINE functions depends on the selected HW platform. Table 1 shows the execution time of the basic operations in SPINE computed on the Telosb platform by the hardware counter connected to the crystal oscillator of the platform.

TABLE I. BASIC OPERATIONS

Operation	Time (ms)
Radio Start-Up	2.685
Radio Shut-Down	0.244
Packet Transmission (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

TABLE II. UTILITIES

Operation	Time (ms)		
	200 elements	100 elements	50 elements
Merge Sort (rec)	23.28	10.62	4.88
Bubble Sort (it)	145.05	36.13	8.88
Max	0.60	0.31	0.18
Mean	0.92	0.55	0.36
Variance	13.67	6.81	3.48

TABLE III. RADIO CONTROLLER AND SPINE PACKETS

Operation	Time (ms)
PacketManager.build – no fragmentation	0.52
PacketManager.build – 2 fragments	1.22
SPINEDataPkt.build	(30bytes of data) 0.092 (200bytes) 0.3
SPINESpineHeader.build	≈ 0.031
SPINESpineHeader.parse	≈ 0.092

TABLE IV. FEATURE EXTRACTORS

Operation	Time (ms)		
	200 elements	100 elements	50 elements
Raw Data (over 3 channels)	0.062		
Max (over 3 channels)	1.67	0.88	0.49
Mean (over 3 channels)	2.68	1.65	1.1
Standard Deviation (over 3 channels)	28.68	17.7	10.8
Vector Magnitude (over 3 channels)	4.58	2.89	2.44
Pitch & Roll (over 3 channels)	19.53	18.37	17.90

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 achievable task allocations that avoid data losses. In particular, since TinyOS is a single task operating system, it does not handle overload gracefully and functionalities must be activated carefully to avoid data loss.

For example, calculation of the standard deviation over 3 channels on a 200 element buffer will take 28.68 msec, implying that if the sensor has a sampling interval lower than 28.68 msec, samples will be lost.

Moreover, depending on the on node feature calculation implementation, there can be a buffer overflow situation when the data in the buffer is ready to be processed before the last computation ends: this implies a lower bound for the sampling rate. In other words, the sensor's sampling rate (Sample_Rate), the number of new elements to wait before a new feature computation (Shift) and the time the feature calculation takes (Time) should be in the following relation:

$$Shift * Sample\_Rate > Time$$

For example, if 200 elements standard deviation must be calculated over 3 channels with a 50% window overlap:

$$\left.\begin{array}{l}\text{Shift} = 50\\ \\ \text{Time} = 28.68\text{ms}\end{array}\right\} \Rightarrow \text{Sample\_Rate}>0.5736\text{ms}$$

Even if this bound is not very strict (sensors we use cannot be sampled so frequently), SPINE users should take care of these results to better design the network and ask nodes to compute functionalities according to their capabilities.

B. Memory Requirements

The data memory has two main components:

- global space (for the global state of the application components)
- stack (for variables locally declared into functions)

Evaluating the memory used by the SPINE Framework requires consideration that significant memory occupancy can reside outside the core, in particular within its extensions (sensor drivers, functions), and is in part influenced by system configuration parameters.

Table 5 shows the memory occupation of the SPINE core functions on a sensor node and a version of SPINE including also Features (Raw Data, Max, Min, Range, Mean, Amplitude, Median, Mode, RMS, Variance, Standard Deviation, Total Multi-Channel Energy, Vector Magnitude, Pitch & Roll) and Alarms (events on Features thresholds), compiled for a Telosb mote platform. Memory analysis should consider the stack growth during the most critical function call chain in the system. Table VI shows the stack occupation of some SPINE commands and events where

$$BUFFERPOOL = sizeof(buffer\_element) \times BUFFER\_POOL\_SIZE \times BUFFER\_LENGTH$$

with:

$$buffer\_element = \#bits\_of\_a\_reading\_to\_be\_stored$$

BUFFER_POOL_SIZE is the total number of circular buffers to be stored into the buffer pool. BUFFER_LENGTH is equal to the buffer size for each buffer of the pool (the buffers are circular, so the buffer size, with the sampling time over a sensor, influences the maximum data time interval storable in that buffer). For example, the default value of 80 samples allows 2 seconds of data stored if the data is sampled every 25ms.

TABLE V. SPINE CORE MEMORY

Description	ROM (bytes)	RAM (bytes)
SPINE 1.2 Core only (TinyOS 2.0.2)	14904	1500 + 2*BUFFER_SIZE* (BUFFERPOOL_SIZE+1)
SPINE 1.2 Core only (TinyOS 2.1)	19812	2062 + 2*BUFFER_SIZE* (BUFFERPOOL_SIZE+1)
SPINE 1.2 with Feature/Alarms on a motion sensorboard (TinyOS 2.0.2)	34316	3860
SPINE 1.2 with Feature/Alarms on a motion sensorboard (TinyOS 2.1)	42768	4736

TABLE VI. STACK SIZE FOR THE MAIN SPINE COMMANDS

Command/Event	Stack (bytes)
PacketManager.build	15 + SPINE_PKT_MAX_SIZE (28)
SensorBoardController. acquisitionDone	69
FunctionManager.send	46
BufferPool.getData	10
FeatureEngine.calculateFeature	25 + $S_{heaviest\_feat\_calculate}$
FeatureEngine. sensorWasSampled	$= BUFFERPOOL + \max(S_{heaviest\_feat\_calculate}, S_{FunctionManager\_send})$
Variance.calculate	17
TotalEnergy.calculate	37
Median.calculate	7 + 510 + $S_{Sort\_mergeSort}$
Mode.calculate	9 + $S_{Sort\_mergeSort}$ + 2 sizeof (dataArray)
PitchRoll.calculate	40 + max(S_{atan2f}, S_{sqrt})
VectorMagnitude.calculate	60

Considering the aforementioned features connected to SPINE, the heaviest function call chain occurs starting from the event handler "sensorWasSampled" of the FeatureEngine component. Here, every shift time, a complete copy of the buffer pool is created; then, taking the data from that copy, all the features requested are computed in a for cycle loop over the list of the user activated features. Finally, the whole computation result buffer is transmitted from the SN to the CN using the generic FunctionManager.send command. The stack occupation for such procedure is:

$$S_{FeatureEngine\_Event\_sensorWasSampled} = BP + \max(S_{heaviest\_feat\_calculate}, S_{FunctionManager\_send})$$

The stack occupation of the heaviest function call chain must be taken into account when customizing SPINE 1.2 to fit into a specific mote platform. For example, MicaZ motes have only 4K of RAM and particular attention must be taken while configuring parameters such the BUFFER_POOL_SIZE (BP) and the BUFFER_LENGTH or while connecting heavy functionalities.

C. Channel Bandwidth

In centralized implementations where raw data is sent from SNs to the CN, the data rate is equal to the sampling rate. In distributed implementations where features are computed on SNs, every node sends a data packet every shift time at the rate:

$$Feature\_Rate = \frac{Sampling\_Rate}{Window * Shift\%}$$

Therefore, in terms of use of the wireless channel bandwidth, transmitting the result of feature computation allows reduction of channel traffic (Window*Shift%) times with respect to transmitting raw data. Table VII reports some examples that show significant savings in terms of pkt/s.

Sampling rate	RawData Rate	Window	Shift%	Feature Rate	% savings
40Hz	40pkt/sec	80samples	50%	1pkt/s	97.5%
40Hz	40pkt/sec	80samples	25%	2pkt/s	95%
10Hz	10pkt/sec	20samples	50%	1pkt/s	90%
10Hz	10pkt/sec	20samples	25%	2pkt/s	80%

TABLE VII. CHANNEL BANDWIDTH USAGE

D. Power Consumption

The radio is one of the most power hungry components of a WSN node and consumes significantly not only when transmits or receives data but also when it listens to the channel waiting for incoming packets. Hence, duty cycling techniques that keep the radio off as long as possible and switch it on when needed are important.

The SPINE Radio Controller component on the node side takes care of switching on the radio when data need to be sent and then manages a listening period to still guarantee a full bidirectional communication with the CN. The application communicates each node whether it should work with the radio always on or not. In case a node is off the CN stores the messages to it until it wakes up. Moreover, an acknowledgment mechanism has been implemented so that the coordinator can ensure the reception of sensor node configuration commands.

This duty cycle technique implemented in SPINE (Fig. 5) allows the node to have the radio off for the most of the time and as a consequence when this mechanism is applied the battery life is greatly increased, as will be shown in the next Section.

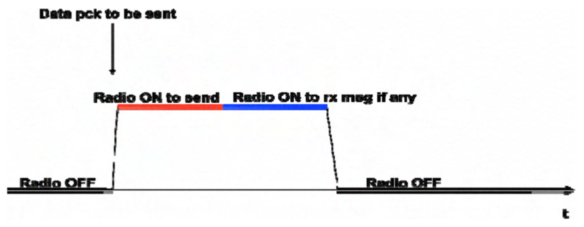

Fig. 5. Radio Controller Duty Cycling

VI. PERFORMANCE ANALYSIS OF A POSTURE RECOGNITION SYSTEM

The SPINE Framework has been used to design an activity recognition system prototype for elderly health monitoring [4]. The prototype is able to recognize postures (e.g. lying, sitting or standing still) and some movements (e.g. walking) of a person and issues an alarm when detects a critical situation, e.g. when the monitored person has fallen.

The sensor node component of the SPINE framework has been implemented on a platform based on Tmote Sky motes with TinyOS execution environment and a custom sensor board (SPINE sensorboard) including a 3-axis accelerometer and two 2-axis gyroscopes. The coordinator component of SPINE has been implemented on a laptop with a Tmote Sky connected via USB port.

The activity recognition system prototype relies on a classifier that takes accelerometer and gyroscope 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, we have selected the K-Nearest Neighbor [10] (KNN) classifier.

The prototype provides a default training set and a graphical interface to let the user build his training set in real time. The significant features to be activated on the node to classify the movements are then selected using the sequential forward floating selection [11] (SFFS) approach or its lightweight version sequential forward selection (SFS). 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, we have selected K=1 and the Manhattan distance as parameters of the classifier.

The experiments have been made using two sensor nodes: one placed on the waist and one on the thigh of the right leg. Then, we have run a SFFS offline implementation to pick the smallest set of features to be activated on the nodes to achieve a sufficiently accurate classification. Online implementation of the selection algorithm might be run as well. In the feature selection algorithm the accuracy has been calculated taking into account half of the training set data to pick the features and half to test the classifier. The identified features are:

- sensor on the waist: mean on the accelerometer axes xyz, min value and max value on the accelerometer axis x;
- sensor on the leg: min value on the accelerometer axis x.

The application has been tested in a lab environment by 4 healthy subjects and soon will be tested in a hospital environment by a larger sample of users.

The performance of the solution has been evaluated in terms of classification accuracy and wireless channel usage. Then the classification algorithm has been tested in a real time deployment of the application. The classification accuracy results are listed in the table below, where data windows have 50% overlap.

TABLE VIII. CLASSIFICATION ACCURACY

Sampling Frequency (Hz)	Data Window (samples)	Accuracy (%)
40	80	96
10	20	98
2	4	88

Although the training set used was obtained by a setting of sampling rate at 40Hz and a data window of 80 samples, with 50% overlapping, the highest accuracy is achieved with different settings. The experiments in table VIII result from subjects whose data was included in the training set. When the same experiment has been made on another subject whose data was not included in the training set, accuracy results slightly changed. For example, in the case of sampling frequency 10Hz and data windows equal to 20 samples, accuracy decreased from 98% to 96%.

The radio optimization of the SPINE framework, together with its on-node signal processing capabilities resulted in a almost 6 times longer battery lifetime period. Battery life is less the 24 hours of continuous monitoring when only raw data readings are transmitted, while it is equal to 5 days if only the most significant features at 1Hz are transmitted. Using an higher capacity battery (during the tests we used a standard camera 3.7V, 600mAh Li-Ion battery) would naturally allow for even longer operation.

These results have been confirmed by an experimental setup as well as by the theoretical calculation of the total battery duration given the battery capacity and the power needed by the sensor every cycle. 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 monitors new accelerometer data and checks if the total energy, calculated on three accelerometer axes, is greater than a threshold (specified at runtime by the user application). Values above the threshold result in an alarm message to the CN. Upon reception of a fall detection message, the CN waits for seven frames from the sensors to evaluate the next seven positions of the person; if it evaluates 4 out of 7 lying positions than an emergency signal is issued.

The client-side application running on the BSN coordinator node, called Physical Activity Monitoring v1.1, allows one to start and control the entire BSN prototype using SPINE Java APIs. In particular, it has four panels: Live Monitoring, Statistics, Advanced and Developers panels. The Live Monitor panel provides a real-time visualization of activity and a log of main SPINE events as they occur. The Statistics panel shows the fraction of time the subject has spent performing each activity. The Advanced panel shows, for each sensor node currently available, its sensor types, available feature extractors, the selected feature extractor, and battery voltage level. The Developers panel is intended for debugging purposes as well as for node functionality tests. It also presents a graph of raw data or feature values as requested, and a log of alarm events.

VII. Conclusions and Future Work

This paper has presented the architecture of SPINE and the implementation parameters that are relevant to design exploration. The latest release of SPINE greatly improves the performance and the flexibility of the framework. In particular, the duty cycling mechanism at the radio controller allows extension of battery lifetime by a factor of six. The definition of a generic sensor interface allows easy introduction of new sensors without affecting the rest of the architecture. We have also described two types of sensor boards currently supported by SPINE, as well as mechanisms to allow designers to specify new services and features have been introduced.

We are currently working on a new release of SPINE with a core in plain C code that can be compiled for several platforms. The aim is to make SPINE able to support multiple SW environments and several existing certified ZigBee platforms.

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. Authors also acknowledge Tekes (Finnish funding agency for technology and innovation).

References

[1] C. Lombriser, D. Roggen, M. Stäger and G. Tröster, “Titan: A Tiny Task Network for Dynamically Reconfigurable Heterogeneous Sensor Networks” In: 15. Fachtagung Kommunikation in Verteilten Systemen (KiVS), pp. 127-138, February 2007.

[2] V. Shnayder, B. Chen, K. Lorincz, T.R.F. Fulford-Jones, and M. Welsh, “Sensor Networks for Medical Care”, Technical Report TR-08-05, Division of Engineering and Applied Sciences, Harvard University, 2005.

[3] Pixie: An Operating System for Resource-Aware Programming of Sensor Networks, http://fiji.eecs.harvard.edu/Pixie

[4] R. Gravina, A. Guerrieri, G. Fortino, F. Bellifemmine, R. Giannantonio, M. Sgroi, “Development of BSN Applications using SPINE”, In 2008 IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC 2008) Singapore, 12-15 October 2008.

[5] SPINE website, http://spine.tilab.com

[6] TinyOS website, http://www.tinyos.net

[7] MoteIV website: http://www.moteiv.com

[8] Shimmer, http://www.shimmer-research.com/

[9] V.-P. Seppä, O. Lahtinen, J. Väisänen, and J. Hyttinen, “Assessment of breathing parameters during running with a wearable bioimpedance device," in Proceedings of the 4th European Congress for Medical and Biomedical Engineering, 2008

[10] T. Cover and P. Hart, “Nearest neighbour pattern classification”, In IEEE Trans. Inform. Theory Vol. 13, pp. 21-27, January 1967.

[11] P. Pudil, J. Novovicova, and J. Kittler, “Floating search methods in feature selection,” Pattern Recognition Letters, 15(11), 1119–1125. Nov. 1994