computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

SPINE-BASED APPLICATION DEVELOPMENT ON HETEROGENEOUS WIRELESS BODY SENSOR NETWORKS

Giancarlo Fortino [1)], Stefano Galzarano [1)], Roberta Giannantonio [2)], Raffaele Gravina [1, 3)], Antonio Guerrieri [1)]

[1)] Dept. of Electronics, Informatics, and Systems (DEIS), University of Calabria, Rende (CS), Italy,
e-mail: g.fortino@unical.it, galzarano@si.deis.unical.it, aguerrieri@deis.unical.it
[2)] TILAB, Telecom Italia, Torino, Italy, e-mail: roberta.giannantonio@telecomitalia.it
[3)] WSN Lab Telecom Italia, Berkeley, CA 94704,
e-mail: rgravina@deis.unical.it

Abstract: *Wireless sensor networks (WSNs) are a novel technology enabling new classes of applications and systems for ubiquitous and pervasive computing. In particular, WSNs for the human body, also known as Wireless Body Sensor Networks (WBSNs), will enable not only continuous, multi-purpose monitoring of people but also will support social interaction among people coming into physical contact. In these contexts, applications demand a wide range of functionalities, in terms of sensor types, processing performance, communication capabilities. Moreover the development of such applications has to deal with the issue of handling heterogeneous WBSNs since different kinds of sensor node architectures could be necessary to fulfill all the application requirements. This paper proposes an approach based on the SPINE frameworks (SPINE1.x and SPINE2) for the programming of signal processing applications on heterogeneous wireless sensor platforms. In particular, two integrable approaches based on the proposed frameworks are described that allow the development of applications for WBSNs constituted by heterogeneous sensor nodes. The approaches are exemplified through a human activity recognition system based on a WBSN composed of two types of sensor nodes, heterogeneous with respect to base software and hardware.*

Keywords: *Wireless body sensor networks, software development methodology, task-oriented programming, distributed signal processing, SPINE.*

1. INTRODUCTION

Wireless body sensor networks (WBSNs) have great potential to enable a broad variety of assisted living applications such as health and activity monitoring, and emergency detection. It is therefore important to provide design methodologies and programming frameworks which enables rapid prototyping of collaborative WBSN applications [1]. Although several effective application development frameworks already exist for WBSNs based on specific sensor platforms (e.g. CodeBlue [2], SPINE [6], Titan [4]), effective methods for platform-independent development of WBSN applications which would enable rapid development of multi-platform applications and fast application porting from one platform to another, are still missing or in their infancy. In fact, the aforementioned frameworks can be only used to effectively develop WBSN applications for TinyOS-based sensor platforms. Thus, to develop applications for new sensor platforms, such frameworks should be implemented for each new sensor platform to be exploited. This not only increases development efforts but also enforces developers to become skilled on the low-level programming abstractions provided by a new employed sensor platform.

In this article we propose two integrable approaches for tackling the development of signal processing applications on heterogeneous WBSNs. Such approaches are based on SPINE and SPINE2, two software frameworks which allow a quick prototyping of WBSN applications. In particular, in the first approach the WBSN coordinator can interact with heterogeneous sensor nodes on which a particular SPINE porting is implemented. In the second approach the WBSN coordinator interacts with heterogeneous nodes each of which is programmed using a same SPINE2 core framework.

The first approach will be presented in the section 2 of this article, while the second approach will be explained in the section 3. In section 4 a real application deployed on a heterogeneous WBSN is described. In the end some brief final considerations will be provided.

2. THE SPINE FRAMEWORK

SPINE (Signal Processing in Node Environment) [3, 6] is a software framework for the design of collaborative Wireless Body Sensor Network (WBSN) applications. It provides programming abstractions, APIs and libraries of protocols, utilities and data processing functions which simplify development of distributed signal processing algorithms for the analysis and the classification of sensor data. SPINE [6] is distributed in Open Source under the LGPL license to facilitate establishing a broad community of users and developers that contribute to the scientific evolution of the framework with new capabilities and applications.

SPINE framework is constituted by two distinctive parts: a node side runtime system residing on the sensor nodes and a Java application, the coordinator, residing on a PC and having functionalities such as nodes configuration and control, data gathering and data analysis.

To date, two releases of SPINE are available:

• The TinyOS release (version 1.3) which supports different kinds of sensor platforms running the TinyOS [7] operating system (supported platforms are TelosB, MicaZ, Shimmer).

• The Z-Stack release (version 1.0) allows the development of WBSN applications on the Z-Stack platform [8] according to the ZigBee standard [9]. In particular, Z-Stack is the implementation of the ZigBee stack carried out by Texas Instruments.

In the following subsections the characteristics of each version of SPINE and the feasibility of integrating both of them into a single heterogeneous WBSN are explained in details.

2.1. SPINE 1.3

The software architecture of the node side part of the framework is reported in Fig. 1. It is composed of a set of nesC components forming the runtime system which relies on the components provided by TinyOS for accessing the hardware resources, such as radio, sensors and timers. More specifically, the *SPINEApplication* is the core of the framework and is responsible for managing the overall system. The *PacketManager* allows the reception/parsing and the formatting/sending of application-level messages over the network through the *RadioController* that, in turn, relies on the communication interfaces (Radio Interface) of TinyOS. The *BufferPool* takes charge of providing a set of buffers in which sensed data and function results are stored. The *SensorBoardController* provides access to the integrated sensors of the node, whereas the *FunctionManager* provides processing capabilities for data pre-elaborations, useful for avoiding battery consumption due to the excessive raw data transmission. This component manages a set of functions already implemented in the release [6], but it is possible to easily extend the framework with other ones, on the basis of the user application requirements.

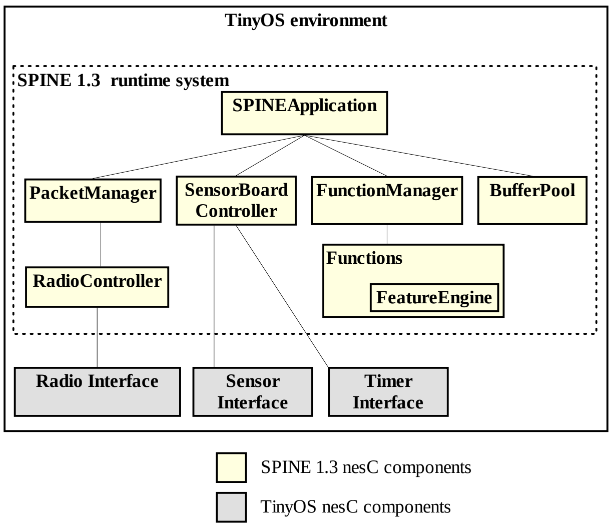


Fig. 1 – The SPINE 1.3 node side framework

```
module FeatureEngineP {  provides {     interface
Function;     interface FeatureEngine;  }
  uses {
   interface Boot;
   interface Feature as Features[uint8_t featureID];
   interface BufferPool;
  }
}
implementation {
  .....
  void calculateFeature(uint8_t featureCode,
                        uint8_t windowSize,
                        uint16_t* bufferPoolCopy) {
   .....
   call Features.calculate[featureCode](
                       (int16_t **)buffer, windowSize);
   result = call Features.getResultSize[featureCode]();
   .....
  }
}
```

Fig. 2 – The *module* code of the SPINE1.3 FeatureEngine component

A particular set of functions handled by the *FunctionManager* is enclosed into the *FeatureEngine* subcomponent. It is in charge of calling appropriate feature calculation on the basis of the computational operations needed on sensed data. A sample code skeleton of the *FeatureEngine* is reported in Fig. 2 and in Fig. 3. Like any other nesC component, the *FeatureEngine* is composed of a “module” and a “configuration” [**7**]. The former acts as a feature dispatcher depending on the *featureCode* parameter while the latter includes all necessary wiring operations for components that actual implement every feature extraction operation.

```
configuration FeatureEngineC {
  provides interface FeatureEngine;
  uses interface Feature
               as Features[uint8_t featureID];
}
implementation {
  components MainC, BufferPoolP, FeatureEngineP;
  // declared feature components
  components MaxC, MinC, RangeC, MeanC, AmplitudeC,
  RmsC, StandardDeviationC, TotalEnergyC, VarianceC,
  ModeC, MedianC, RawDataC, PitchRollC,
  VectorMagnitudeC;

  FeatureEngineP.FeatureEngine = FeatureEngine;
  FeatureEngineP.BufferPool -> BufferPoolP;
  FeatureEngineP.Boot -> MainC.Boot;
  FeatureEngineP.Features = Features;

  // wiring of the declared components
  FeatureEngineP.Features[MAX] -> MaxC;
  FeatureEngineP.Features[MIN] -> MinC;
  FeatureEngineP.Features[RANGE] -> RangeC;
  FeatureEngineP.Features[MEAN] -> MeanC;
  FeatureEngineP.Features[AMPLITUDE] -> AmplitudeC;
  FeatureEngineP.Features[RMS] -> RmsC;
  FeatureEngineP.Features[ST_DEV] ->
                              StandardDeviationC;
  FeatureEngineP.Features[TOTAL_ENERGY] ->
                              TotalEnergyC;
  FeatureEngineP.Features[VARIANCE] -> VarianceC;
  FeatureEngineP.Features[MODE] -> ModeC;
  FeatureEngineP.Features[MEDIAN] -> MedianC;
  FeatureEngineP.Features[RAW_DATA] -> RawDataC;
  FeatureEngineP.Features[PITCH_ROLL] -> PitchRollC;
  FeatureEngineP.Features[VECTOR_MAGNITUDE] ->
                              VectorMagnitudeC;
}
```

Fig. 3 – The *configuration* code of the SPINE1.3 FeatureEngine component

2.2. SPINE FOR Z-STACK

The framework architecture of the SPINE for Z-Stack is shown in Fig. 4. It consists of components having different names from the ones of the version 1.3, but having same functionalities because, obviously, the framework has to offer the same interface to the high level application running on top of it.

The implementation of these components, in C language, is dependent on the services provided by the Z-Stack system [8]. One of its parts is the Operating System Application Layer (*OSAL*) which is not, strictly speaking, considered an Operating System, because of its very limited functionality. However, it offers APIs for: tasks managing (initialization, scheduling, synchronization and message passing), interrupts handling, timers managing and memory allocation. All these functionalities are represented in Fig. 4 as different blocks: *Task Creation System*, *Task Synch System*, *Timers* and *Memory Management*.

Every Z-Stack application has to be defined as a set of tasks, each of which has to implement proper routines for initialization and management of events, like timeouts, interrupts or incoming messages from other tasks. So, the application lifecycle depends on how tasks are configured and how tasks interact with each others through messages exchange.

All the framework components are implemented as tasks running upon the OSAL infrastructure.

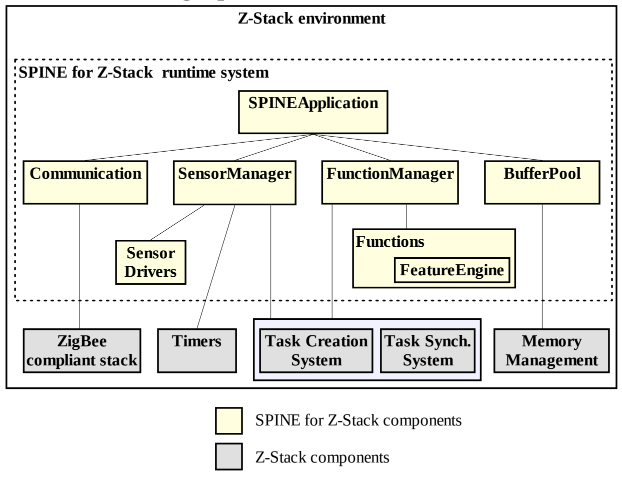


Fig. 4 – The SPINE for Z-Stack node side framework

The *SensorManager* relies on the *Task Creation System* for the creation of the necessary tasks responsible for sensing operations.

A limitation of the Z-Stack environment is that it supports the creation of tasks only at the system start-up, but this is not compliant to the framework which should manage sensing tasks at runtime. So, it was necessary to directly modify a part of the Z-Stack specification, for having the possibility to add new tasks on running system.

Moreover, the stack does not offer APIs for accessing physical sensor, simply because the board comes without installed sensors. For this reason, an accelerometer has been interfaced to the microcontroller and a proper driver has been implemented for providing a direct access to it. In the future other physical sensors and relative drivers could be added to the framework. The *SensorManager* is responsible for managing sensor drivers and associates them to the sensing tasks.

Every acquisition operation is associated to a sampling time and for this purpose the *Timers* component offers a simple way for associating a particular timeout event of a timer to the execution of a task. It suffices that a sensing task calls the timer allocation function and the OSAL will be in charge of signaling, with an event, the execution of the task after the timer expiration. This approach differs from the one adopted in SPINE1.3, where an explicit association to the timer and the consequent sensing operation execution compete to the programmer. The part of the framework that substantially differs in the two versions is related to the communication management. In fact, in order to be executed on the Z-Stack environment, the framework has to be ZigBee compliant. So, all the application-level messages have to be encapsulated in a message packet formatted according to the protocol stack specifications. This is performed through the use of APIs provided by the *ZigBee Device Object* of the Z-Stack system.

The *FunctionManager* manages the processing functions on the node and makes use of the *Task Synch System* to request result transmission to an appropriate communication task. The *BufferPool* has the same functionality as the *BufferPool* of SPINE1.3. As in SPINE1.3, the *FunctionManager* handles the *FeatureEngine* subcomponent for sensed data pre-elaboration. Its sample C code is shown in Fig. 5. Differently from the SPINE1.3 version of the *FeatureEngine*, the version for Z-Stack consists of a series of explicit case statements depending on the unique code of the requesting feature.

2.3. HETEROGENEOUS PROGRAMMING

The SPINE coordinator, which runs at the base station side, is able to interact both with TinyOS sensor nodes and Z-Stack sensor nodes (as shown in Fig. 6). This allows building a WBSN composed of heterogeneous nodes which however should be programmed by using the node side SPINE implementation for each specific node. So, while this approach allows using heterogeneous sensors in the same WBSN, different types of sensors must be differently programmed. In order to perform its functionalities, the coordinator has to be interfaced, via USB cable, to one of each sensor type (TinyOS and Z-Stack) using the appropriate radio communication capabilities for communicating with two different parts of the WBSN (it is worth noting that TinyOS sensor platforms supported by SPINE 1.3 use the IEEE 802.15.4 standard while Z-Stack uses the ZigBee standard). Nevertheless the high-level communication service is the same as it uses a unique format for application-level messages used for nodes configuration and information exchange.

Furthermore, the coordinator can be either local or remote; in fact, a new implementation of the coordinator supports multi-user remote application control through RMI technology. When it is needed to use different types of sensors within a WBSN application, it is important to know how much the framework performs differently on them. A performance comparison between SPINE 1.3 on TelosB nodes and SPINE for Z-Stack nodes, concerning the processing of specific features computed on different sizes (50, 100) of data samples, is reported in Table 1.

```
uint8 featureEngine_calculateFeature(
              functionParameter_t* par,
              active_feature* activeFeatureList,
              uint8 actFeatsIndex, FEspace* workspace){
 .....
   callFeature(
        activeFeatureList[actFeatsIndex].FeatureCode,
        par->ch, activeFeatureList[actFeatsIndex].mask,
        par->windowSize, result);
 .....
   buildResult(workspace->result, result,
          activeFeatureList[actFeatsIndex].FeatureCode,
          activeFeatureList[actFeatsIndex].mask);
 .....
}

uint8 callFeature(FeatureCodes code, int16** ch,
           uint8 mask, uint8 length, uint16** result){

 uint8 op_result = OP_ERROR;

 switch(code){
  case MAX:{
   op_result= feature_Max(ch,mask,length,result);
   break;
  }
  case MIN:{
   op_result= feature_Min(ch,mask,length,result);
   break;
  }
  case RAW_DATA:{
   op_result=feature_RawData(ch,mask,length,result);
   break;
  }
  case MEAN:{
   op_result=feature_Mean(ch,mask,length,result);
   break;
  }
  case AMPLITUDE:{
   op_result=feature_Amplitude(ch,mask,length,result)
   break;
  }
  case RMS:{
   op_result= feature_Rms(ch,mask,length,result);
   break;
  }
  case ST_DEV:{
    op_result=feature_StandardDeviation(ch,mask,length,
                                          result);
    break;
  }
  case VARIANCE:{
    op_result=feature_Variance(ch,mask,length,result);
    break;
  }
  case MEDIAN:{
   op_result=feature_Median(ch,mask,length,result);
   break;
  }
  case PITCH_ROLL:{
    op_result=feature_PitchRoll(ch,mask,length,result);
     break;
  }
  case VECTOR_MAGNITUDE:{
    op_result=feature_VectorMagnitude(ch,mask,length,
                                        result);
     break;
  }
  case MODE:{
    op_result=feature_Mode(ch,mask,length,result);
    break;
  }
  case TOTAL_ENERGY:{
    op_result=feature_TotalEnergy(ch,mask,length,
                                  result);
    break;
  }
  default:break;
 }
 return op_result;
}
```

Fig. 5 – The code of the FeatureEngine component of SPINE for Z-Stack

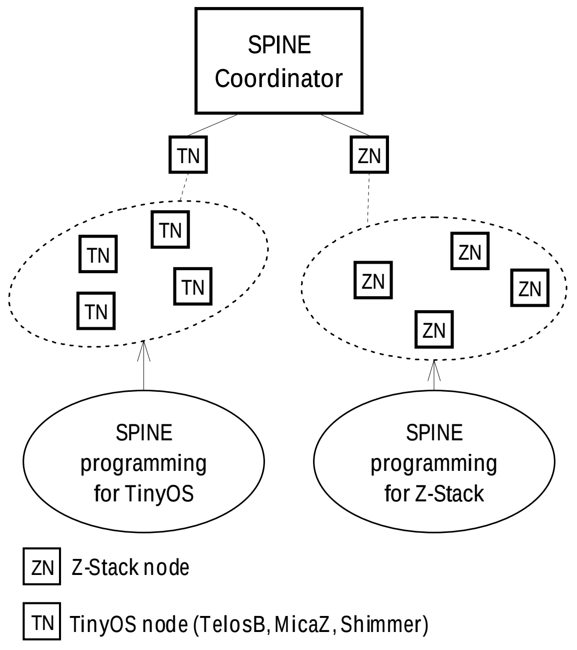

SPINE Coordinator
TN
ZN
TN
ZN
SPINE programming for TinyOS
SPINE programming for Z-Stack
ZN Z-Stack node
TN TinyOS node (TelosB, MicaZ, Shimmer)

Fig. 6 – A SPINE heterogeneous network

As can be noted, a Z-Stack node provides better performance than TelosB (obviously, because of its different and higher performance hardware), so it is more desirable to use Z-Stack nodes to accomplish complex and time-consuming processing task.

Table 1. Performance comparison between SPINE 1.3 (on TelosB sensor nodes) and SPINE for Z-Stack

Feature	Time (ms)			
	100 samples		*50 samples*	
	TelosB	*Z-Stack*	*TelosB*	*Z-Stack*
Max	0.88	0.27	0.49	0.15
Mean	1.65	0.53	1.1	0.32
Vector Magnitude	2.89	1.29	2.44	0.76
Pitch Roll	18.37	1.13	17.9	0.85
Standard Deviation	17.7	0.96	10.8	0.55

3. THE SPINE2 FRAMEWORK

The SPINE2 framework [10] is an evolution of SPINE based on the C-language for reaching a very high platform independency for C-like programmable sensor platforms (e.g. TinyOS, Ember [11], Z-Stack) and so raising the level of the provided programming abstractions from platform-specific to platform-independent.

To develop platform-independent WSN applications, several approaches, defined for platform-independent software development in conventional distributed platforms, can be effectively adopted:

- *Model-driven Development (MDD)*. MDD is an approach which provides a set of guidelines for structuring specifications expressed as models and, then, translating such models into platform-dependent code [12]. In particular, MDD defines system functionality using a platform-independent model (PIM) through an appropriate domain-specific language (DSL); then, given a platform definition model (PDM) corresponding to CORBA, .NET, the Web, etc., the PIM is transformed into one or more platform-specific models (PSMs) that computers can run. The PSM may use different DSLs or general purpose languages (e.g. Java, C#, PHP, Python). Moreover, automated tools generally perform this transformation.
- *Virtual Machine (VM)*. A VM runs as a normal application inside an OS. Its purpose is to provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform.
- *Software Layering (SL)*. Software layering has been largely used for the development of communication protocol suites to hide network heterogeneity. Therefore to hide heterogeneity of different platforms a basic software layer (or core framework), which provides basic functionality, is defined for a set of heterogeneous platforms based on a similar programming language and adapted to each different platforms through platform specific modules. Code development is carried out through such common programming language according to the defined core framework.

Although MDD approach is very flexible and effective for platform-independent software development, a major problem is that automatic translation may introduce overhead in terms of generated code size and execution speed. The VM approach is effective for providing platform independence, but the deployment of a VM on node able to execute the SPINE language can be very expensive in terms of execution speed and used resources (e.g. memory). According to the SL approach, a SPINE2 core framework can be defined through a language used by the majority of sensor platform and, then, adapted to such different platforms through platform specific software modules. With this approach the core framework can be accurately defined and implemented and kept highly efficient. However, it fits only sensor platforms programmed through compatible languages.

SPINE2 was founded on an SL approach based on the C language (see Fig. 7), which is the language used for programming the majority of embedded systems. It embodies the following features:

- execution on commercial resource-constrained sensor platforms each one having a different operating system;
- minimization of the amount of code that should be replicated for each specific implementation;
- enabling C-developers (eventually C++) to extend the SPINE2 framework without having to

learn low-level details of specific sensor platforms or without having to learn new programming languages;
• enabling compiling and simulating the code by using normal ANSI C tools.

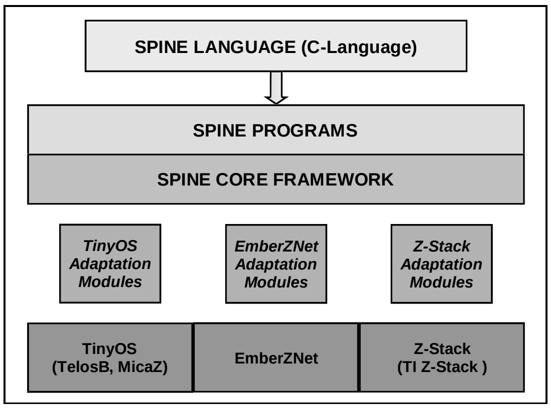

SPINE LANGUAGE (C-Language)
SPINE PROGRAMS
SPINE CORE FRAMEWORK
TinyOS Adaptation Modules
EmberZNet Adaptation Modules
Z-Stack Adaptation Modules
TinyOS (TelosB, MicaZ)
EmberZNet
Z-Stack (TI Z-Stack)

Fig. 7 – SPINE2 based on the SL approach

While SPINE is centered on a programming model based on functions, SPINE2 is based on a task oriented programming model in order to best fit the requirements of collaborative distributed applications in resource-constrained environments. Distributed and collaborative applications can then be programmed as a dynamically schedulable and reconfigurable set of tasks. Different tasks can be assigned to each node of the network and tasks can be controlled at execution time via proper message exchange; in this way the network can overall adapt to changes in context, in overall goals, in the state of each single node, and it can better balance load and task types between each element of the network. Dynamic distribution of tasks also allows preprocessing of sensed data directly on the node, a significant reduction of data transmission and battery consumption, and an overall increase of the network lifetime. Thanks to task oriented programming, application developers do not need to program in tiny environments but only configure tasks on the WBSN coordinator.

Fig. 8 shows the software architecture of SPINE2. Basically, it consists of the similar functional parts previously discussed for the versions 1.x, but it is important to observe that, according to the adopted SL approach, most of the framework components are implemented in C. Focusing attention on the changed parts, because of the new task oriented programming model, the *TaskManager* takes the place of the *FunctionManager* (see Fig. 1 and Fig. 4). Moreover it relies on the platform-specific *TaskScheduler* which provides scheduling functionalities for the tasks activated on the node. The *BufferPool* allows storage capabilities for sensed data and task results. It is an active component because it consists not only of a set of buffers, but provides a simple API in order to safely access them. Finally, due to its programming model, the communication protocol has been redesigned and new message types have been introduced so that to allow task management on SPINE2 nodes, such as task creation and configuration. Nevertheless, to maintain backward compatibility, a SPINE1.x/2 software communication bridge has been also implemented.

It is quite clear that if programmers want to extend the SPINE2 framework for supporting other C-like sensor platforms, the only components that they have to develop are few adapter modules that interface the SPINE2 core (the C modules) to the platform-specific hardware resources, such as radio, sensors and timers.

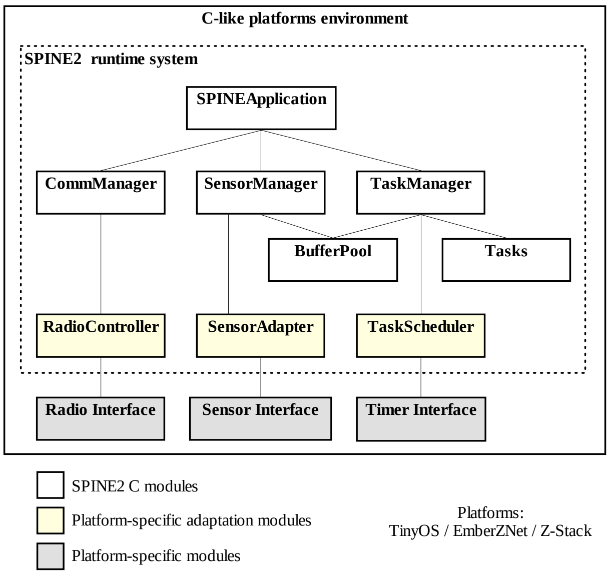


Fig. 8 – The SPINE2 node side framework

Currently, the program execution control implemented in SPINE2 is based on a timer-driven approach [13]. Differently from others approaches, like the data-flow-driven or the event-driven, each single task composing a user application is activated by a timer. Therefore, the developer not only has to establish relationships among tasks, but also set their timers in a properly way, such that they are kept synchronized. For example, a data manipulation task working on a buffered sensed data should be associated to a timer so that it fires only when all data are available, and this depends on settings of the timer related to the task associated to the sensing operation.

According to the new programming model a timed task is defined as a C-struct as reported in Fig. 9. In particular, *taskID* is a unique identifier, *taskType* is the type of the task, *status* holds information about the task status (created, active, paused), *timer* contains the task firing time, *timerScale* contains the measurement unit of the timer, *isPeriodic* signals if the timed task is periodic or one-shot, and *parameters* contains parameters specific to the *taskType*. The currently available taskTypes are *sensing*, *featureExtraction*, and *aggregation&sending*.

```
typedef struct timedTaskDescription {
unsigned char taskID;  unsigned char
taskType;  unsigned char status;  unsigned
long timer;
  unsigned char timerScale;
  unsigned char isPeriodic;
  parameters[TASK_PARAMETER_LENGTH];
}timedTaskDescription;
```

Fig. 9 – The SPINE2 timed task definition

Fig. 10 shows a SPINE2 net formed by different sensor platforms. According to this new framework, not only the SPINE coordinator can interact with heterogeneous nodes, but also developers can program nodes in homogeneous way.

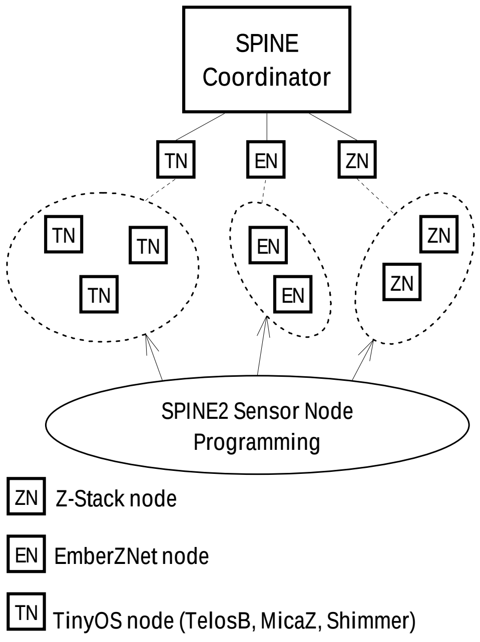

SPINE Coordinator
TN
EN
ZN
SPINE2 Sensor Node Programming
ZN Z-Stack node
EN EmberZNet node
TN TinyOS node (TelosB, MicaZ, Shimmer)

Fig. 10 – A SPINE2 heterogeneous network

In fact, the node side SPINE2 core is implemented through a high-level programming approach and this code is the same for each platform supported by SPINE2. This also allows a simpler and more rapid approach (compared to version 1.x) for framework extension, for example when a new sensor or a new functionality is to be added because of user application needs.

After having described the new characteristics of version 2 of the SPINE framework, an analysis of its performance compared to the performances of the previous versions is presented to show which implications the use of the Software Layering approach involves. In particular, time performance evaluations related to the execution of specific features on the two different types of nodes are reported in Table 2. As one can see, the SPINE2 software architecture does not induce any sort of performance penalties but, on the contrary, exhibits small improvements. Furthermore, in the Z-Stack environment there are no differences between SPINE and SPINE2, because in both versions, calls to functions are based on the C language.

Table 2. Performance comparison of features extraction evaluated on 100 data samples among SPINE (1.x) and SPINE2

Feature	Time (ms)			
	SPINE		SPINE2	
	TelosB	*Z-Stack*	*TelosB*	*Z-Stack*
Max	0.88	0.27	0.86	0.27
Mean	1.65	0.53	1.56	0.53
Vector Magnitude	2.89	1.29	2.50	1.29
Pitch Roll	18.37	1.13	13.82	1.13
Standard Deviation	17.7	0.96	17.43	0.96

4. ACTIVITY MONITORING BASED ON HETEROGENEOUS WBSNs

To test the effectiveness of the SPINE (versions 1.x and 2) frameworks and their capability on managing a heterogeneous network, a human Activity Monitoring System (AMS) [3] has been reverse engineered and made heterogeneous.

AMS 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 or unable to stand up. Fig. 11 shows the entire software design of the system, which consists of a coordinator-side application and two node-side applications.

The former is implemented in Java (in Fig. 11 is represented as “Application on SPINE”) and contains classifiers that use the gathered pre-elaborated data coming from the sensors, for performing recognition of movements and postures defined in a training phase. This application runs on top of the SPINE Manager, enclosed in the coordinator part of the SPINE framework. This manager can exploit distinctive communication modules for interacting with different types of sensor communication protocols.

The node-side application is deployed on two sensor nodes, one located on the thigh and the other on the waist of the person, and running on top of the SPINE 1.x/2 node runtime system. Both of them rely on sensed data coming from an accelerometer, but no raw data are sent to the coordinator because they are pre-elaborated by some processing functions (features extraction) before their transmission. The two sensor node applications differ regarding the way sensor data are pre-processed and transmitted. In particular, the thigh node application consists of sending to the coordinator the result of the *Min* feature extracted from the X-axis data of the accelerometer. The application residing on the waist node takes data from all the three axes of the accelerometer and split them for the computation of three different features: the *Mean* from all axes, the *Min* and the *Max* from only the X-axis. Afterwards, all the computed features are aggregated together and sent to the coordinator.

Each feature has two main parameters that have to be set for a correct application definition: *window* and *shift*. The window value represents the number of sampled data on which the feature is evaluated, whereas the shift value represents how many new samples data are necessary for a new computation of the same feature.

It is very important to consider that the application design in Fig. 11 has been represented following the SPINE2 task-oriented approach, but the application specifications are not, however, strictly feasible only for an implementation on the SPINE2 node runtime system. In fact, the AMS application has been implemented and tested, separately, both with the SPINE1.x and with the SPINE2 framework.

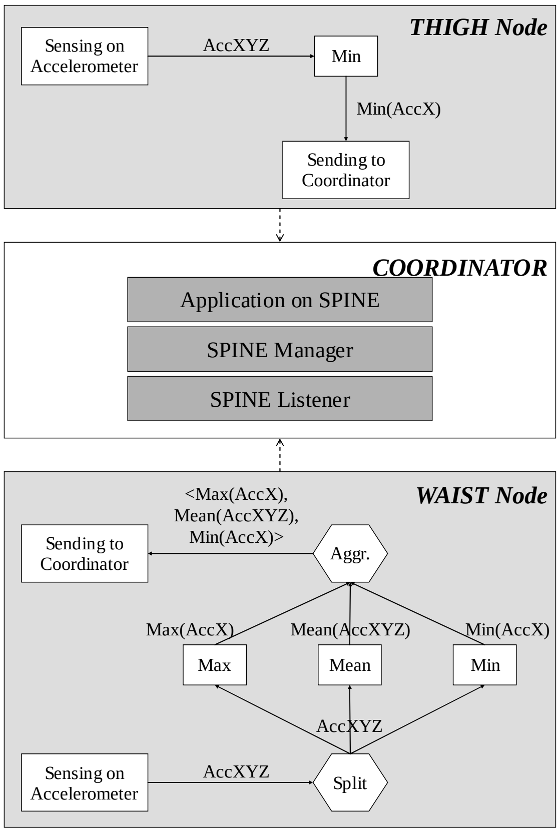


Fig. 11 – The Activity Monitoring application

In Fig. 12 the complete SPINE 1.x Activity Monitoring System is shown. The network architecture is composed of a TelosB node (running the SPINE 1.3 runtime) placed on a thigh of a person and a Z-Stack node (running the SPINE for Z-Stack runtime) placed on the waist of the same person. In section 2.3 we have already discussed about the possibility of having different types of sensor in the same SPINE net and this real application demonstrates the feasibility in managing a heterogeneous WBSN using the SPINE1.x framework. The coordinator (running on a notebook) has been interfaced with the WSN through other two nodes connected via USB cable which provide the necessary radio communication capability.

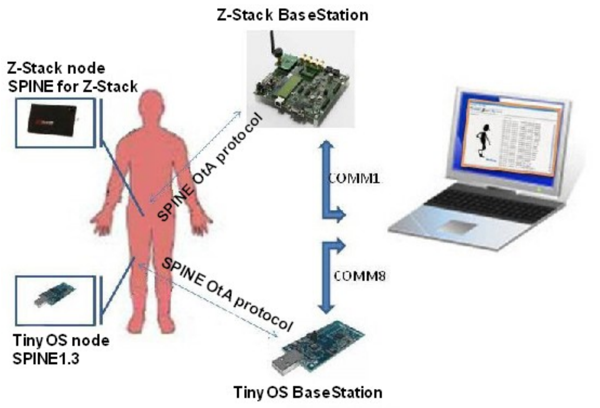


Fig. 12 – The SPINE1.x Activity Monitoring System

As mentioned before, the system has also been tested with the version 2 of the SPINE framework. The system architecture, depicted in Fig. 13, is the same as the previous except that in such a case, the different types of nodes run the same SPINE2 core runtime system plus the particular adaptation code related to the particular node platforms. Moreover, the SPINE2 Over-the-Air communication protocol is necessary because of the new task oriented programming model adopted in SPINE2.

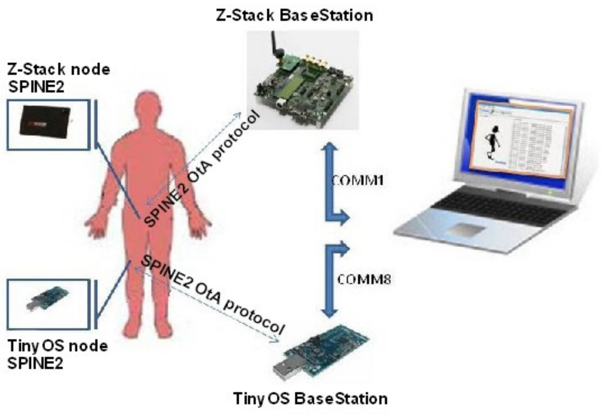


Fig. 13 – The SPINE2 Activity Monitoring System

To show the SPINE2 programming of the sensors, the waist node application (see Fig. 11) is considered. The definition of the timed tasks (see Fig. 9) is reported in Fig. 14. For the other two feature extraction tasks (Max and Min), parameters settings are similar to the Mean task, except for the FEX_FEATURE and the FEX_CHANNEL_BITMASK which is 0x08, indicating that only data from axis X of the accelerometer have to be considered. The timed tasks on the thigh node are defined in a similar way.

```
(sensTask)->taskID = 1;
(sensTask)->taskType = TASKTYPE_SENSING;
(sensTask)->timer = 50;
(sensTask)->timerScale = TIMER_SCALE_MSEC;
(sensTask)->isPeriodic = TRUE;
(sensTask)->parameters[ACQ_SENSOR_ID] = ACCELEROMETER;
(sensTask)->parameters[ACQ_CHANNEL_BITMASK] = 0x0E;
                                        //XYZ channels
(sensTask)->parameters[ACQ_BUFFER_ID_1] = 0;
(sensTask)->parameters[ACQ_BUFFER_ID_2] = 1;
(sensTask)->parameters[ACQ_BUFFER_ID_3] = 2;
```

(a)

```
(meanTask)->taskID = 2;
(meanTask)->taskType = TASKTYPE_FEATURE_EXTRACTION;
(meanTask)->timer = 500; //every 10 new samples
(meanTask)->timerScale = TIMER_SCALE_MSEC;
(meanTask)->isPeriodic = TRUE;
(meanTask)->parameters[FEX_FEATURE] = MEAN;
(meanTask)->parameters[FEX_CHANNEL_BITMASK] = 0x0E;
(meanTask)->parameters[FEX_WINDOW] = 20;
(meanTask)->parameters[FEX_BUFFER_ID_1] = 0;
(meanTask)->parameters[FEX_BUFFER_ID_2] = 1;
(meanTask)->parameters[FEX_BUFFER_ID_3] = 2;
(meanTask)->parameters[FEX_SENSOR_ID]=ACCELEROMETER;
(meanTask)->parameters[FEX_AGGR_ID] = 1;
```

(b)

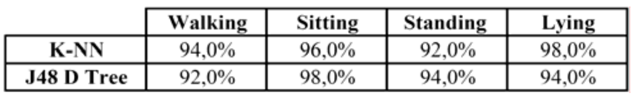
```
(aggrSendTask)->taskID = 5;
(aggrSendTask)->taskType = TASKTYPE_AGGR_AND_SEND;
(aggrSendTask)->timer = 550;
(aggrSendTask)->timerScale = TIMER_SCALE_MSEC;
(aggrSendTask)->isPeriodic = FALSE;
(aggrSendTask)->parameters[AGG_ID] = 1;
(aggrSendTask)->parameters[AGG_FEATURES_TO_WAIT_FOR]=3;
```

(c)

Fig. 14 – The definition of tasks for the waist sensor node: (a) sensing task, (b) feature extraction (Mean) task, (c) aggregation&sending task.

The application on the coordinator is responsible for gathering pre-elaborated data taken from the accelerometer sensors of the nodes and relies on a classifier that recognizes postures and movements defined in a training phase. In particular, the application integrates two different classifiers: one based on the K-Nearest Neighbor algorithm [14] and the other based on J48 Decision Tree [15]. They were setup through a training phase and tested considering the following settings for the sensors data acquisition: the sample time was set to 50ms, the window to 20, whereas the shift to 10. This means that the features in Fig. 11 (Min, Max and Mean) are evaluated on 20 sampled data (1 sec acquisition) and computed every new 10 samples (500ms) acquired by the sensors. See Table 3 for the obtained classification accuracy results.

Table 3. Classification accuracy for classifiers based on K-Nearest Neighbor and J48 Decision Tree



	Walking	Sitting	Standing	Lying
K-NN	94,0%	96,0%	92,0%	98,0%
J48 D Tree	92,0%	98,0%	94,0%	94,0%

5. CONCLUSION

In the context of the rapid development of WBSN applications, this paper has introduced the main features of the SPINE and SPINE2 frameworks. They can be effectively used for enabling the development of signal processing applications on heterogeneous WBSNs. In particular, the approach based on SPINE relies on the capability of the WBSN coordinator to interact with a network composed of heterogeneous sensor nodes on which a platform specific porting of the SPINE framework is installed. In the second approach based on SPINE2, the WBSN coordinator is still able to interact with heterogeneous nodes but, in this case, on each sensor node the same SPINE2 core is installed so allowing sensor node homogeneous programming. Results obtained from the performance evaluation of the SPINE frameworks show that SPINE2 performs better than SPINE1.3 on TelosB sensor nodes in terms of speed for feature computation.

On-going work aims to: (i) complete the implementation of SPINE2 for the Ember sensor platform and designing a version for ContikiOS [16], (ii) develop a SPINE2 coordinator based on a task-oriented protocol to program and control SPINE2 sensor nodes, (iii) design a flexible event-based architecture for SPINE2 to increase programming effectiveness and avoid an excessive use of timers, and (iv) extend SPINE2 for general collaborative WSN applications (not only centered on star-based networks).

6. ACKNOWLEDGMENTS

Authors wish to thank Luigi Buondonno and Antonio Giordano for their implementation efforts on porting SPINE on the Z-Stack platform and Fabio L. Bellifemine and Marco Sgroi for their precious contributions to the SPINE project in terms of ideas, discussions and useful suggestions. The SPINE project is partially funded by Telecom Italia.

7. REFERENCES

[1] O. Gama, C. Figueiredo, P. Carvalho, P. M. Mendes. Towards a Reconfigurable Wireless Sensor Network for Biomedical Applications. *IEEE International Conference on Sensor Technologies and Applications (SensorComm)*, Valencia (Spain), 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] R. Gravina, A. Guerrieri, G. Fortino, F.

Bellifemine, R. Giannantonio, M. Sgroi. Development of body sensor network applications using SPINE. *In Proc. of IEEE International Conference on "Systems, Man, and Cybernetics (SMC2008)*, Singapore, Oct. 12-15, 2008.

[4] C. Lombriser, N.B. Bharatula, D. Roggen. On-body activity recognition in a dynamic sensor network. *In Proc. of 2nd Int. Conference on Body Area Networks (BodyNets 2007)*, Florence, Italy, June 11-13 2007.

[5] S. Iyengar, F. Tempia Bonda, R. Gravina, A. Guerrieri, G. Fortino, A. Sangiovanni-Vincentelli. A framework for creating healthcare monitoring applications using wireless body sensor networks. *In the Proc. of the 3rd International Conference on Body Area Networks (BodyNets'08)*, Tempe (AZ), USA, Mar. 13-15, 2008.

[6] SPINE documents and software. http://spine.tilab.com

[7] TinyOS Web Site. http://www.tinyos.net

[8] Z-Stack – ZigBee Protocol Stack – http://focus.ti.com/docs/toolsw/folders/print/z-stack.html

[9] ZigBee Alliance – http://www.zigbee.org/

[10] Giancarlo Fortino, Antonio Guerrieri, Fabio Bellifemine, Roberta Giannantonio. Platform-independent development of collaborative Wireless Body Sensor Network applications: SPINE2. *In Proc. of 2009 IEEE International Conference on Systems, Man, and Cybernetics (SMC2009)*, San Antonio (TX) USA, Oct. 11-14, 2009.

[11] Ember Web Site. http://www.ember.com

[12] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, vol. 20, no. 5, pp. 19-25, Sep./Oct. 2003.

[13] G. Fortino, A. Guerrieri, R. Giannantonio, F. Bellifemine. SPINE2: developing BSN applications on heterogeneous sensor nodes. *In Proc. of IEEE Symposium on Industrial Embedded Systems (SIES'09), special session on wireless health*, Lausanne (Switzerland), 8-10 July 2009.

[14] T. Cover, P. Hart. Nearest neighbor pattern classification. *In IEEE Trans. Inform. Theory*, Vol. 13, pp. 21-27, January 1967.

[15] R. Quinlan. *C4.5: Programs for Machine Learninge*. Morgan Kaufmann Publishers. San Mako, CA, 1993.

[16] Contiki, documentation and software http://www.sics.se/contiki.

__Giancarlo Fortino__, is an Associate Professor of computer science at the Department of Electronics, Informatics, and Systems of the University of Calabria, Italy. His research interests include distributed computing, wireless sensor networks, multimedia systems, agent-oriented technology and systems, and applied software engineering. He received a Laurea degree and a PhD in Computer Engineering from the University of Calabria.

__Stefano Galzarano__, is a Master Student in Computer Engineering at the University of Calabria. His research interests are focused on high-level programming methods for wireless sensor networks. He received a Bachelor degree in Computer Engineering from the University of Calabria.

__Roberta Giannantonio__, is a Researcher at the Telecom Italia Lab, Torino, Italy. Her research interests are mainly focused on wireless technology, particularly wireless sensor networks. She received a Laurea degree in Telecommunication Engineering from Politecnico di Torino.

__Raffaele Gravina__, is a PhD Student in Computer Engineering at the University of Calabria. His research interests are focused on high-level programming methods for wireless sensor networks. He received a Bachelor and Master degrees in Computer Engineering from the University of Calabria.

__Antonio Guerrieri__, is a PhD Student in Computer Engineering at the University of Calabria. His research interests are focused on high-level programming methods for wireless sensor networks. He received a Bachelor and Master degrees in Computer Engineering from the University of Calabria.