Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Platform-independent development of collaborative Wireless Body Sensor Network applications: SPINE2

G. Fortino, A. Guerrieri
Dept. of Electronics, Informatics and Systems (DEIS)
University of Calabria
Via P. Bucci, 87036 Rende (CS), Italy
g.fortino@unical.it, aguerrieri@deis.unical.it

F. Bellifemine, R. Giannantonio
Telecom Italia Labs
Via G. Reiss Romoli, 274. Torino, Italy
{fabioluigi.bellifemine,
roberta.giannantonio}@telecomitalia.it

***Abstract*—Rapid development of Wireless Body Sensor Network (WBSN) applications can be enabled by suitable domain-specific frameworks which are usually organized in two parts: a base-station-side (or coordinator) and a sensor-node-side. While the former can be based on the Java language so being highly portable, the latter is usually highly dependent on the exploited sensor platform. Available state of the art frameworks follow such an organization and, in particular, the current version of SPINE is based on TinyOS and can be only used to effectively develop collaborative WBSN applications for TinyOS-based sensor platforms. To develop SPINE-based applications for new sensor platforms, the SPINE framework should be re-implemented for each new sensor platform to be exploited. This not only increases development efforts but also enforces SPINE-oriented developers to become skilled on the low-level programming abstractions provided by a new employed sensor platform. In this paper we discuss issues related to platform-independent development of collaborative WBSN applications and, specifically, describe the requirements, architecture and first implementation experiences of SPINE2 which aims at reaching a very high platform independency and raising the level of the used programming abstractions by providing a task-oriented programming model. The paper also discusses how such a task-oriented model enables dynamic task assignment and holistic collaborative task execution also for resource-constrained environments such as tiny sensor nodes.**

***Keywords*—Wireless body sensor networks, software development methodology, task-oriented programming.**

I. 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 [3], 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 paper we first categorize and discuss interesting approaches which can effectively support platform-independent development of WSN applications (see section 2); then (in section 3 and 4), we present the current efforts (design and current implementation status) towards the definition of SPINE2, an evolution of the SPINE (Signal Processing In-Node Environment) framework [3] based on the C-language, which aims at supporting the development of platform-independent assisted living applications. The goal is to reach a very high platform independency for C-like programmable sensor platforms (e.g. TinyOS [5], Ember [6], ZStack [7]) and raise the level of the provided programming abstractions from platform-specific to platform-independent.

SPINE2 offers a task-oriented model for programming the sensor nodes of a collaborative WBSN. In particular tasks (e.g. sensing, feature extraction, aggregation, data transmission) can be dynamically discovered, created, activated, scheduled and controlled by the coordinator on each sensor node in order to fulfill a goal-directed overall task of the distributed system implemented by the network of sensor nodes. Dynamic distribution of tasks allows – among the others – 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. Different tasks can be assigned to each node and tasks can be controlled at execution time via proper message exchange; in this way the network can overall adapt to changes in context, overall goals, state of each single node, and it can better balance load and task types between each element of the network. Such a task-oriented model enables a holistic approach where the WBSN capability becomes higher than the sum of the capabilities of each element.

The novelty of this paper is the discussion of how such a model was implemented in resource-constrained environments, and what architecture and approach was selected in order to achieve platform independency and provide high level programming abstractions that reduce time-to-market for tiny environments.

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

II. PLATFORM-INDEPENDENT DEVELOPMENT OF WSN APPLICATIONS

To develop platform-independent WSN applications several approaches, defined for platform-independent software development in conventional distributed platforms, can be effectively adopted, as described in the followings.

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 [8]. 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 a general purpose language like Java, C#, PHP, Python, etc. Moreover, automated tools generally perform this transformation. In [9], an MDD framework to manage the complexity of application development for WSNs is proposed. This framework consists of a UML profile for WSN applications and a UML virtual machine, named Matilda. The proposed UML profile abstracts the low-level details of WSNs and provides higher abstractions for application developers to graphically design and maintain their applications. Matilda is a runtime engine used to design, validate, deploy and execute WSN applications consistently at the modeling layer. In [10], it is argued that current software development of wireless sensor networks not only imposes much work on low-level programmers but also prevents domain experts from directly contributing parts of the software. The proposed solution is based on the exploitation of domain-specific languages which are inexpensive to define, have a syntax that domain experts understand, and creating simulations for them is easy. An application modeled through a domain-specific language could be translated into (low-level) platform-dependent code or into bytecode for a virtual machine (see VM approach below). Finally in [11], a framework based on Simulink, Stateflow and Embedded Coder which follows the MDD approach is presented. By using such framework, an engineer can create sensor network components (both at the application and at the protocol level) that can be used as building blocks to model, simulate and automatically generate code for different underlying platforms and operating systems (TinyOS and MantisOS).

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. Several efforts have been devoted to the definition of VMs for the programming of WSN application (Matè, Deluge, SOS, Agilla, etc). In particular, Maté [12] is a byte code interpreter (VM) running on TinyOS, that provides safe program execution environments, runtime re-programming, and an event-driven stack-based architecture. Applications running in Maté use instructions that are interpreted by virtual processors programmed onto network nodes. The performance penalty of the interpretation of the instructions can be alleviated by adding application-specific instructions to the virtual machine. Maté also supports dynamic reconfigurability of nodes through code diffusion.

*Software Layering (*SL). Software layering has been largely used for the development of communication protocol suites to hide network heterogeneity; TCP/IP is the most notable example. Therefore to hide heterogeneity of different sensor platforms a basic software layer (or core framework), which provides basic functionality, is defined for a set of sensor platforms based on a similar programming language and adapted to each different sensor platform through platform specific modules. Code development is carried out through such common programming language according to the defined core framework. In the context of WSNs, this approach is still scarcely used; however, due to its specific characteristics, it is adopted for the definition of SPINE2 (see next section)

III. SPINE2

SPINE (Signal Processing in Node Environment) [3, 13] 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 1.0 was developed in the TinyOS environment (node side) and in Java language (server side); it is distributed [14] 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. The latest release is SPINE 1.2 that, on the node side, supports different kinds of hardware platforms running the TinyOS operating system.

The subject of this paper is the new on-going developments to release version 2 of SPINE which aims to become independent on the low level details and operating systems of the used sensor platform. In order to fulfill this goal, each of the approaches introduced in section 2 might be used.

According to the MDD approach (see Fig. 1) platform independent models developed through the SPINE language, defined as a DSL, are translated into platform specific code through platform specific translators. Although this 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.

According to the VM approach (see Fig. 2) a SPINE VM programmable through a SPINE programming language should be defined and implemented for each sensor platform to be used. Although this approach is effective for providing platform independence, 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 (see Fig. 3) a SPINE core framework is 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.

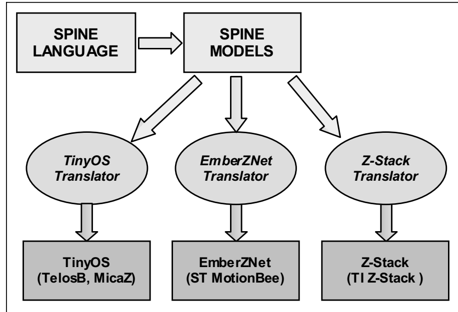

Figure 1. SPINE2 based on the MDD approach.

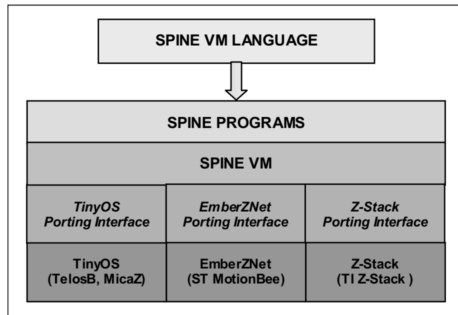

Figure 2. SPINE2 based on the VM approach.

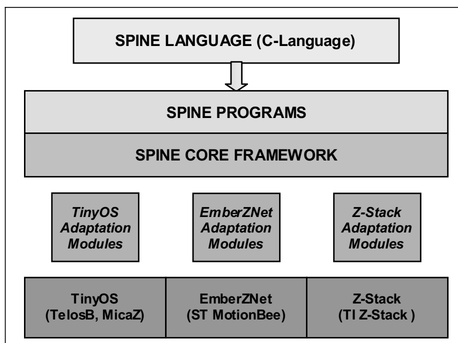

Figure 3. SPINE2 based on the SL approach.

Considered that our requirements were the followings:

- execution on commercial resource-constrained sensor platforms, such as TinyOS [5], Ember [6] or Texas [7], 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 SPINE framework without having to learn low-level details of specific sensor platforms or without having to learn new programming languages, such as nesC [5];
- enabling compiling and simulating the code by using normal ANSI C tools;

we selected the SL approach founded on the C language, which is the language used by the sensor platforms we selected and, as a matter of fact, by the majority of resource-constrained environments. In the next three subsections the programming model, the architecture and the communication protocol of SPINE2 are described.

A. *Programming Model*

In SPINE2 a task-oriented programming model was implemented on the nodes in order to best fit the requirements of collaborative distributed applications in a resource-constrained environment: an agent is executed on each sensor node that – via proper message exchange – can discover, create, activate, schedule and control tasks. 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 allows – among the others – 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.

Application developers do not need to program in tiny environments. A SPINE agent is installed on each node that allows interacting with the base station for the task assignment and control. An application is simply a recipe listing the set of tasks to be assigned to each node; the recipe can be executed via the Java API of the base station and, via the same API, the recipe can also be changed at execution time in order to implement different states and intentions of the system.

So far, the following types of tasks have been implemented:

- *SensingTask*, which allows defining a sensing operation on a given sensor. The sensing operation can be one-shot or periodic.
- *TimingTask*, which allows defining timers for timing other tasks. Timers can be one shot or periodic.
- *FunctionalTask*, which refers to the functional tasks defined through programming:
  - *ProcessingTask*, which allows to elaborate data. A specific type of processing is the feature extraction (or simply feature), a data processing algorithm which is carried out on a set of values that can be taken from a data buffer of the BufferPool.
  - *AggregationTask*, which allows aggregating data calculated by different functions.
  - *TransmissionTask*, which allows transmitting data produced by sensing, processing and/or aggregation tasks.

An example of task-oriented programming is shown in Fig. 4 by means of a data-flow-based model. In this example, the sensed data generated by the *Sensing* task (by acquiring data from a 3-axis accelerometer) are fed to the *Split* task that, in turn, splits the data for the computation of three features. Each

feature is implemented as a Task and fed with different data: the *Mean* task uses data from all three axes (XYZ), the *Min* and *Max* tasks uses data from the X axis. Each triple of computed features (<Mean(AccXYZ), Min(AccX), Max(AccX)>) are aggregated by the *Aggregation* task (Aggr) and sent to the destination node by the data transmission task (*Sender*). The reader can easily guess the variety of complex tasks which can be created by using such a task composition formalism.

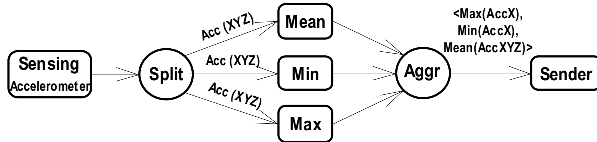

Figure 4. Data-flow-based model.

The SPINE task-oriented programming model is not data-flow driven but event-driven. Thus, the data-flow model of Fig. 4, which can be seen as a model at a higher-level of abstraction, can be defined according to the event-driven model of SPINE2 as in Fig. 5. In particular, data, which are sensed by the Sensing task driven by a Timer1 set to a given sampling rate, are stored into decoupling buffers for the three axes channels (AccX, AccY, AccZ). Buffers are managed by the BufferPoolManager component which is based on the event-based publish subscribe paradigm. It accepts subscriptions coming from the FunctionTasks and as soon as events related to such subscriptions occur, it notifies, through the AcqNotify, the FunctionTask subscriber which, in turn, can fetch the data which it is interested in. In the proposed example, the ProcessingTasks are notified by the BufferPoolManager when S sensed data samples have been acquired; where S is the shift parameter of the ProcessingTasks which compute their function on a sample window (W) equals to n*S samples. The processed data are passed to the AggregrationTask (Aggr) which, after aggregation, passes them to the TransmissionTask (Sender). In Fig. 6 a timer-driven model of the same example is shown. It simplifies the architecture supporting SPINE2 models by avoiding the introduction of the BufferPoolManager active component. In particular, the FunctionTasks are not driven by events sent by the BufferManagerPool but by timers appositely set upon the timer driving the SensingTask. Timer2, Timer3, and Timer4 are therefore set to S*Timer1. Moreover, also the Aggr&SendTask (a task which jointly aggregates and transmits) is timer driven. Timer5=Timer2+0.1*Timer2.

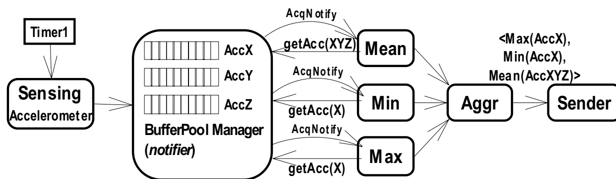

Figure 5. Event-driven SPINE2-based model.

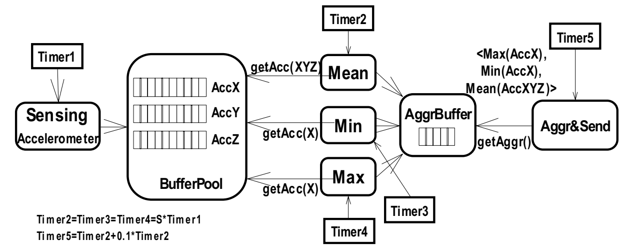

Figure 6. Timer-driven SPINE2-based model.

B. A Timer-driven Architecture

The timer-driven architecture of SPINE2 (Fig. 7) consists of a SPINE core framework which is to be adapted to platform-specific components (sensor drivers, application lifecycle, timers, communication). The SPINE core framework currently implements the task execution logics according to the timer-driven programming model.

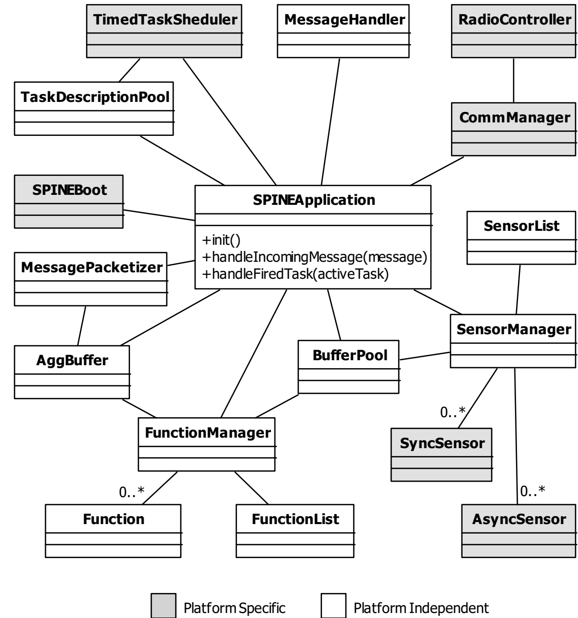

Figure 7. The SPINE2 component diagram.

The platform-independent components of the SPINE2 framework are:

- The SPINEApplication is the core component of a SPINE2 application. It reacts to external events, like messages, and to internal events. It provides three functions:
  - *init* for application initialization;
  - *handleIncomingMessage* for handling an incoming network message;
  - *handleFiredTask* for handling fired tasks.
- The TaskDescriptionPool component is a dynamic list of the tasks created in the node.

- The BufferPool component consists of a set of data buffer needed to store data produced by sensing tasks and aggregate data computed by functions.
- The FunctionManager component, which acts as a Dispatcher managing the available list of functions (FunctionList), is called by the SPINEApplication to execute ProcessingTasks, AggregationTasks and TransmissionTasks.
- The AggrBuffer component, which allows to temporary store computed features for aggregating them.
- The SensorManager component is connected to platform specific components, which are specific drivers for the Sensors components. The SensorManager manages a list of sensors (SensorList) which can be of synchronous and asynchronous type. While SynchronousSensors are read through a synchronous primitive, AsynchronousSensors are based on (i) an asynchronous primitive for requesting a sensor read and (ii) the related notification of the read value, which is done by the sensor driver, when data is available, so that the sensor manager can fetch it.
- The MessageHandler component contains the handling code of the SPINE2 protocol packets.
- The MessagePacketizer component allows building packets according to the SPINE2 protocol.

The aforementioned platform-dependent components are to be appositely adapted to the following platform-dependent components which drive their execution:

- The SPINEBoot component is the application entry-point which provides the application lifecycle. It contains platform specific initializations and wirings, and drives the SPINEApplication by calling its init() method.
- The TimedTaskScheduler component manages timed tasks by using platform-specific timers.
- The AsyncSensor/SyncSensor components are the sensor drivers, which can be synchronous or asynchronous, through which real sensors can be accessed.
- The CommManager component contains the platform-specific radio communication logic. CommManager can receive messages from the RadioController and pass them to the SPINEApplication through the handleIncomingMessage method which uses the MessageHandler component. In particular, the RadioController allows to handle the sensor radio for receiving and transmitting packets and for putting the radio in stand-by for energy saving.

The SPINE 1.2 communication protocol which enables communication between the base station and the sensor node is function-oriented. As the programming model of SPINE 2.0 is task oriented, the SPINE2 communication protocol was re-designed. Nevertheless, to maintain backward compatibility, a SPINE1.2/SPINE2 software communication bridge was also implemented. The SPINE2 communication protocol is task-oriented and, in particular, provides the following packet types:

- *createTask*, which allows to create a task with the associated parameters;
- *startTask*, which starts a created task or restart a paused task;
- *pauseTask*, which pauses a started task;
- *updateTask*, which reconfigures a paused task;
- *deleteTask*, which stops and/or cancel a task;
- *getTasksDescription*, which returns the list of the created tasks of the node and their status;
- *startNode*, which starts all the tasks created on the node;
- *getNodeConfiguration*, which returns the configuration of the node in terms of available sensors, functions and tasks;
- *data*, which contains data sent from the node to the coordinator.

IV. IMPLEMENTATION STATUS

SPINE2.0 architecture as described in Figure 7 easily allows supporting different platforms since the platform specific components are well defined. The parts to be implemented include the application lifecycle, the communication part, the timer related stuff and the low level access, that are all the OS specific parts as previously described. SPINE2.0 currently supports two software sensor platforms, TinyOS2.1 and Texas Instruments Z-Stack1.2.

Although these platforms are all based on a C-like programming language, they differ not only in terms of operating systems but also in terms of programming abstractions and communication protocols.

While TinyOS was designed as a general purpose open source operating system for wireless sensor networks, Texas Instrument Z-Stack is a software platform certified to be ZigBee compliant. On the other hand, while being more flexible in terms of applications and modules to be inserted, TinyOS brings another complexity due to the programming paradigm and language. Therefore such two platforms are very good candidates for evaluating the flexibility of the SPINE2.0 architecture.

Z-Stack environment defines not only an operating system but also standard related logics that must be preserved to maintain the standard compliance. ZigBee standard [15] defines not only the low level communication layers (IEEE 802.15.4 as MAC protocol, ZigBee network, management and security layers) and standard application profiles (such as the Home Automation one) but also basic rules to be followed when building a proprietary profile.

As an application level framework, SPINE2.0 on TinyOS has been implemented as an application module whereas on the Z-Stack platform has been designed to be a proprietary ZigBee profile with respect to all the ZigBee related rules.

In particular, the parts to be added into the different platforms are the followings:

- *SPINEBoot* takes care of the system initialization:

  - in the Z-Stack it defines all the ZigBee standard descriptors as well as sets up the ZigBee communication part;

  - in TinyOS this is simply the application entry point from where the TinyOS application is compiled and started and does not include any logic rather than the SPINE related one.

• *CommManager/RadioController* takes care of all the communication related operations:

- in the Z-Stack this part takes care of the communication through primitives defined by the standard for the application protocol data units transport between peer application entities (ZigBee APS Data Service).

- in TinyOS it uses send/receive low level APIs for communicating with other devices in the network.

• *TimedTaskScheduler* schedules the timed tasks created in the sensor node:

- in the Z-Stack implementation this part takes care of all the timed event through an extensive usage of the utilities provided by the Z-Stack OS (OSAL, Operating System Abstraction Layer) such as task allocation, timer settings and so on. It is important to notice that SPINE2.0 task oriented architecture run without any modification even into a task oriented OS as the OSAL is;

- the timer interface already defined in the OS is used in the TinyOS version to manage timer related events

• *Sensor drivers* which are platform specific since they have to access to low level functionalities.

Moreover each platform will need specific configuration files setting all the tunable parameters.

The Timer-driven SPINE2.0 release has been successfully implemented and tested on both mentioned platforms. In the following we elucidate the structure and programming of the available tasks.

A timed task is defined as a C-struct as follows:

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

where, *taskID* is the unique task identifier, *taskType* is the type of 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*, *feature extraction*, and *aggregation & sending*:

```
enum taskTypes{
 TASKTYPE_FEATURE_EXTRACTION = 0x01,
 TASKTYPE_SENSING = 0x02,
 TASKTYPE_AGGR_AND_SEND = 0x03
};
```

In particular the parameters of task types are defined as follows:

```
enum sensing_TaskType{
 SENS_SENSOR_ID = 0,      //id of the sensor
 SENS_CHANNEL_BITMASK = 1,//bitmask for Ch selection
 SENS_BUFFER_ID_1 = 2,    //buffer associated to Ch1
 SENS_BUFFER_ID_2 = 3,    //buffer associated to Ch2
 SENS_BUFFER_ID_3 = 4,    //buffer associated to Ch3
 SENS_BUFFER_ID_4 = 5     //buffer associated to Ch4
};
```

```
enum feature_extraction_TaskType{
 FEX_FEATURE = 0,        //id of the feature
 FEX_CHANNEL_BITMASK = 1,//bitmask for Ch selection
 FEX_WINDOW = 2,         //data window
 FEX_BUFFER_ID_1 = 3,    //buffer associated to Ch1
 FEX_BUFFER_ID_2 = 4,    //buffer associated to Ch2
 FEX_BUFFER_ID_3 = 5,    //buffer associated to Ch3
 FEX_BUFFER_ID_4 = 6,    //buffer associated to Ch4
 FEX_SENSOR_ID = 7,      //id of the sensed sensor
 FEX_AGGR_ID = 8         //id of the aggr&send task
};

enum aggregation_and_sending_TaskType{
 AGG_ID = 0,        //id of the aggr&send task
 AGG_FEATURES_TO_WAIT_FOR = 1, // aggr feature number
 AGG_TIMER = 2,      //reference aggr timer
 AGG_DEF_COUNTER = 6 //number of aggr trials
}
```

The example of Fig. 6 is implemented and successfully tested on TelosB motes [16] and TI ZStack sensor nodes both equipped with specific 3-axial accelerometer sensor boards. In particular the defined timed tasks (sensing from accelerometer, calculation of mean, and aggregation and sending) are the following:

• TASKTYPE_SENSING

```
(sensTask)->taskID = 1
(sensTask)->taskType = TASKTYPE_SENSING
(sensTask)->timer = 25
(sensTask)->timerScale = 1 //ms
(sensTask)->isPeriodic = 1 //true
(sensTask)->parameters[ACQ_SENSOR_ID]=1 //accelerometer
(sensTask)->parameters[ACQ_CHANNEL_BITMASK]=e//Ch XYZ
(sensTask)->parameters[ACQ_BUFFER_ID_1] = 0
(sensTask)->parameters[ACQ_BUFFER_ID_2] = 1
(sensTask)->parameters[ACQ_BUFFER_ID_3] = 2
```

• TASKTYPE_FEATURE_EXTRACTION_1

```
(featExtTask)->taskID = 2
(featExtTask)->taskType = TASKTYPE_FEATURE_EXTRACTION
(featExtTask)->timer = 1000 //40 samples
(featExtTask)->timerScale = 1
(featExtTask)->isPeriodic = 1
(featExtTask)->parameters[FEX_FEATURE] = 5 //MEAN
(featExtTask)->parameters[FEX_CHANNEL_BITMASK] = e
(featExtTask)->parameters[FEX_WINDOW] = 80
(featExtTask)->parameters[FEX_BUFFER_ID_1] = 0
(featExtTask)->parameters[FEX_BUFFER_ID_2] = 1
(featExtTask)->parameters[FEX_BUFFER_ID_3] = 2
(featExtTask)->parameters[FEX_SENSOR_ID] = 1
(featExtTask)->parameters[FEX_AGGR_ID]=1
```

• TASKTYPE_AGGR_AND_SEND

```
(aggrSendTask)->taskID = 5;
(aggrSendTask)->taskType = TASKTYPE_AGGR_AND_SEND;
(aggrSendTask)->timer = 1100;
(aggrSendTask)->timerScale = TIMER_SCALE_MSEC;
(aggrSendTask)->isPeriodic = FALSE;
(aggrSendTask)->parameters[AGG_ID] = 1;
(aggrSendTask)->parameters[AGG_FEATURES_TO_WAIT_FOR]=3;
(aggrSendTask)->parameters[AGG_TIMER] = 1000;
(aggrSendTask)->parameters[AGG_DEF_COUNTER] = 0;
```

Tasks can be created and started either by explicit node programming or by the base station through the SPINE2 protocol.

To experiment with SPINE2, the sensor-node side application presented in [3], which allows the activity monitoring of individuals (standing, lying, walking and sitting), is now based on sensor nodes supported by SPINE2.0.

Experimentation with SPINE2 was also carried out to demonstrate that the platform independency of SPINE2 does not introduce performance penalties with respect to SPINE1.2.

To this purpose, an evaluation of the data processing performances of the TinyOS versions of SPINE1.2 and SPINE2 on the TMote SKY TelosB sensor platform [16] has been carried out. The performance evaluation results are reported in Table I. In particular, selected features (max, mean, standard deviation, vector magnitude, pitch & roll, and entropy) were computed on different sample sizes (50, 100, 200) acquired from a 3-axial accelerometer sensor board. As can be noted, SPINE2 feature processing performances are higher than those computed with SPINE1.2. This performance improvement is mainly due to the different methods of calling processing functions in SPINE1.2 and SPINE2: in SPINE1.2 a processing function is executed by means of a call to a nesC command whereas in SPINE2 a simple call to a C function of an included file is done.

TABLE I. COMPARISON OF FEATURE PROCESSING TIMES (MS) THROUGH SPINE1.2 AND SPINE2 IN TINYOS ON TELOSB SENSOR NODES

	50 samples		100 samples		200 samples	
	SPINE1.2	SPINE2	SPINE1.2	SPINE2	SPINE1.2	SPINE2
MAX	0,488	0,488	0,883	0,886	1,696	1,679
MEAN	1,069	1,038	1,627	1,556	2,714	2,625
STANDARD DEVIATION	10,070	7,450	16,703	13,823	28,617	26,054
VECTOR MAGNITUDE	2,136	1,741	3,126	2,501	4,564	3,997
PITCH & ROLL	17,894	16,967	18,401	17,428	19,461	18,552
ENTROPY	239,291	235,848	488,185	481,167	1016,392	1001,735

V. CONCLUSIONS AND FUTURE WORK

In this paper we have presented SPINE2 a domain-specific framework for the platform-independent development of collaborative WBSN applications. SPINE2 consists of two parts: (i) the core which is written in C and is independent from any C-like sensor platform on which it can be ported on; (ii) a set of platform-dependent components (sensors, communications, timers, application lifecycle) through which the core can be easily adapted. The task-oriented programming model of SPINE2 enables a flexible development of WBSN applications in terms of a star-based network of collaborative and dynamically reconfigurable tasks which concur to carry out an overall distributed task. SPINE2 (specifically the timer-driven architecture) is currently implemented for TinyOS sensor platforms (in particular for TelosB motes) and ZStack Zigbee sensor nodes, and was successfully applied to the realization of a multi-platform WBSN application for activity monitoring of individuals. Moreover, SPINE2 does not only introduce performance penalties but also increases performances of feature calculation with respect to SPINE1.2. On-going work is geared at (i) completing the implementation of SPINE2 for the Ember sensor platform and designing a version for ContikiOS; (ii) designing and implementing a flexible event-based architecture for SPINE2, and (iii) extending SPINE2 for more general collaborative WSN applications (not only centered on star-based networks).

ACKNOWLEDGMENTS

Authors wish to thank Luigi Buondonno and Antonio Giordano for the implementation of SPINE2 on ZStack, Mostafiz Rahman Mozumdar for the on-going implementation of SPINE2 on Ember, and Raffaele Gravina and Marco Sgroi for useful suggestions about the SPINE2 project. 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] 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 (SMC 2008), 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] TinyOS Web Site. http://www.tinyos.net

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

[7] ZStack website.http://focus.ti.com/docs/toolsw/folders/print/z-stack.html

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

[9] H. Wada, P. Boonma, J. Suzuki, and K. Oba, "Modeling and executing adaptive sensor network applications with the Matilda UML Virtual Machine," In Proc. of the 11th IASTED International Conference on Software Engineering and Applications (SEA), Cambridge, MA, November 2007.

[10] D. A. Sadilek, “Prototyping domain-specific languages for wireless sensor networks,” In Proc. of the 4th International Workshop on Software Language Engineering, 2007.

[11] M. M. Rahman Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, S. Olivieri, “A framework for modeling, simulation and automatic code generation of sensor network applications,” In Proc. of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks, San Francisco (CA), USA, 2008.

[12] P. Levis and D. Culler, “Mate: a Virtual Machine for Tiny Networked Sensors,” In Proc. of ASPLOS, Dec 2002.

[13] 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.

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

[15] ZigBee Alliance. http://www.zigbee.org/

[16] Tmote SKY TelosB. http://www.sentilla.com/moteiv-transition.html