Advance Access publication on February 22, 2010
doi:10.1093/comjnl/bxq019

A Java-Based Agent Platform for Programming Wireless Sensor Networks†

FRANCESCO AIELLO, GIANCARLO FORTINO*, RAFFAELE GRAVINA AND ANTONIO GUERRIERI

Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Via P. Bucci, cubo 41c, 87036 Rende (CS), Italy
**Corresponding author: g.fortino@unical.it*

Wireless sensor networks (WSNs) are emerging as powerful platforms for distributed embedded computing supporting a variety of high-impact applications. However, programming WSN applications is a complex task that requires suitable paradigms and technologies capable of supporting the specific characteristics of such networks which uniquely integrate distributed sensing, computation and communication. Mobile agents are a distributed computing paradigm based on code mobility that has already demonstrated high effectiveness and efficiency in IP-based highly dynamic distributed environments. Due to their intrinsic characteristics, mobile agents may provide more benefits in the context of WSNs than in conventional distributed environments. In this paper we present the design, implementation and experimentation of MAPS (Mobile Agent Platform for Sun SPOT), an innovative Java-based framework for wireless sensor networks based on Sun SPOT technology which enables agent-oriented programming of WSN applications. The MAPS architecture is based on components that interact through events. Each component offers a minimal set of services to mobile agents that are modeled as multi-plane state machines driven by ECA rules. In particular, the offered services include message transmission, agent creation, agent cloning, agent migration, timer handling and easy access to the sensor node resources (sensors, actuators, input switches, flash memory and battery). Agent programming with MAPS is presented through both a simple example related to mobile agent-based monitoring of a sensor node and a more complex case study for real-time human activity monitoring based on wireless body sensor networks. Moreover, a performance evaluation of MAPS carried out by computing micro-benchmarks, related to agent communication, creation and migration, is illustrated.

Keywords: agent systems; mobile agents; event- and state-based programming; wireless sensor networks; body sensor networks; human activity monitoring

Received 4 November 2009; revised 4 December 2009
Handling editor: Alex Rogers

1. INTRODUCTION

Due to recent advances in electronics and communication technologies, wireless sensor networks (WSNs) have been introduced and are currently emerging as one of the most disruptive technologies enabling and supporting next generation ubiquitous and pervasive computing scenarios. WSNs have a high potential to support a variety of high-impact applications such as disaster/crime prevention and military applications, environmental applications, health-care applications and smart spaces. However, programming WSNs is a complex task due to the limited capabilities (processing, memory and transmission range) and energy resources of each sensor node as well as the lack of reliability of the radio channel. Moreover, WSN programming is usually application-specific (or more generally domain-specific) and requires tradeoffs in terms of

†Revised and extended version of the paper: Aiello, F., Fortino, G., Gravina, R. and Guerrieri, A. (2009) MAPS: A Mobile Agent Platform for Java Sun SPOTs. *Proc. 3rd Int. Workshop on Agent Technology for Sensor Networks* (*ATSN-09*), jointly held with the *8th Int. Joint Conf. Autonomous Agents and Multiagent Systems* (*AAMAS-09*), May 12, Budapest, Hungary.

task complexity, resource usage and communication patterns. Therefore the developed software, which usually integrates routing mechanisms, time synchronization, node localization and data aggregation, is tightly dependent on the specific application and scarcely reusable. Thus to support rapid development and deployment of WSN applications flexible, WSN-aware programming paradigms are needed, which directly provide proactive and on-demand code deployment at runtime as well as ease software programming at the application, middleware and network layers.

Among the programming paradigms proposed for the development of WSN applications [1, 2], the mobile agent-based paradigm [3, 4], which has already demonstrated its effectiveness in conventional distributed systems as well as in highly dynamic distributed environments, can effectively deal with the programming issues that WSNs have posed. In particular, a mobile agent is a software entity encapsulating dynamic behavior and able to migrate from one computing node to another to fulfill distributed tasks. We believe that mobile agents can provide more benefits in the context of WSNs than in conventional distributed environments. In particular, mobile agents can support the programming of WSNs at the application, middleware and network levels. At the application level, mobile agents can be used as design and programming abstractions through which WSN applications can be effectively designed and implemented. At the middleware level, mobile agents can be used for implementing WSN core services such as data aggregation/fusion/dissemination and query-based information retrieval, and for dynamically deploying new services through efficient code dissemination. At the network level, mobile agents can be used as the mobile capsules in active networks for smart multi-hop routing and other network services. A few trials have to date been devoted to the development of mobile agent systems (MASs) for wireless sensor networks (Agilla [5], actorNet [6], SensorWare [7]); however, none of them has been specifically developed for the Sun SPOT sensor platform [8], which is completely programmable in Java, supported by the SquawkVM [9] and compatible with J2ME. Indeed, a noteworthy agent-based framework for J2ME devices, agent factory micro edition (AFME) [10], has been recently ported on Sun SPOT; however, not being conceived specifically for Sun SPOT technology, AFME does not completely exploit its functionality.

In this paper, we propose MAPS (Mobile Agent Platform for Sun SPOT), an innovative Java-based framework for wireless sensor networks based on Sun SPOT technology which enables agent-oriented programming of WSN applications. The architecture of MAPS is component-based and offers a minimal set of services to mobile agents, including message transmission, agent creation, agent cloning, agent migration, timer handling and easy access to the sensor node resources (sensors, actuators, input switches, flash memory and battery). The dynamic behavior of mobile agents is modeled as multi-plane ECA-based state machines. MAPS therefore enables a highly effective application programming through an integration of three of the most important paradigms for WSN programming: agent-oriented, event-based and state-based programming. The effectiveness of MAPS is demonstrated by the development of simple example applications (e.g. mobile agent-based remote monitoring of sensors) as well as a more complex case study. While the former aims at testing MAPS and describing how to program an application with MAPS, the latter highlights the effectiveness and suitability of MAPS for developing a real-time system supporting human activity monitoring which is of enormous importance in the health-care domain. However, great effectiveness usually implies performance penalties; to address such issues a performance evaluation of the basic mechanisms of MAPS and Sun SPOT (e.g. communication and migration) has been carefully carried out to quantify the unavoidable overhead introduced by MAPS.

The contributions that this work offers to the WSN programming research area are the following:

(i) A novel Java-based agent platform for Sun SPOT that allows an effective Java-based development of agents and agent-based applications by integrating agent-oriented, event-driven and state-based programming paradigms. Moreover, being based on a component-based approach, which clearly separates component interfaces from their implementations, MAPS is easily portable on other Java sensor platforms (e.g. Sentilla JCreate sensors [11]).

(ii) The performance evaluation carried out allows evaluating not only MAPS per se but also the degree of maturity of the Sun SPOT technology for supporting (mobile) agent-based applications and systems. An interesting result is that migration is still an open issue since Sun SPOT mechanisms supporting migration still need to be improved (e.g. lack of dynamic class loading) and optimized (hibernation and serialization of Isolates are too time-consuming operations).

(iii) The development of a real case study concerned with testing the effectiveness and the suitability of the agent-based approach featured by MAPS for the development of a complex application in the health-care domain such as the real-time monitoring of human activities.

The rest of the paper is organized as follows. Section 2 discusses related work and, in particular, currently available (mobile) agent systems/platforms for WSNs. Section 3 presents the requirements, architecture and the agent programming model of MAPS. Section 4 describes the implementation of MAPS based on the Java Sun SPOT library. In Section 5, a simple example is provided for exemplifying the agent-based application programming with MAPS. Section 6 shows the performance evaluation of MAPS carried out through micro-benchmarks, whereas Section 7 proposes a real case study developed through MAPS for the real-time monitoring of human activities based on wireless body sensor networks

(WBSNs). Finally, conclusions are drawn and future work briefly anticipated.

2. RELATED WORK

Mobile agents are supported by MASs, which basically provide an agent server, an Application Programming Interface (API) for mobile agent programming and, sometimes, supporting programming and administration tools. In particular, the agent server is able to execute agents by providing them with basic services such as migration, communication and resource access. In the last decade, a significant number of MASs for IP-based distributed computing systems have been developed [3]. The majority of them are Java-based (e.g. Aglets, Voyager, Ajanta, JADE etc.) and few others rely on other languages (D'Agents, ARA etc.).

In the context of WSNs it is challenging to develop MASs for supporting mobile agent-based programming [12]. Due to the currently available resource-constrained sensor nodes and related operating systems, building flexible and efficient MASs is a very complex task. Very few MASs for WSNs have to date been proposed and actually implemented. The most significant ones are: SensorWare [7], Agilla [5] and actorNet [6]. A general mobile-agent-oriented sensor node architecture to which such MASs adhere is shown in Fig. 1. The MAS relies on the services offered by the OS and the mobile agents are executed within the MAS, which supports their inter-node migrations, sensing capabilities and resource access, and inter-agent communications.

SensorWare [7] is a general middleware framework based on agent technology, where the mobile agent concept is exploited. Mobile control scripts in Tcl model network participants' functionalities and behaviors, and routing mechanisms to destination areas. Agents migrate to destination areas performing data aggregation reliably. The script can be very complex and diffusion gets slower when it reaches destination areas. The replication and migration of such scripts in several sensor nodes allows the dynamic deployment of distributed algorithms into the network. SensorWare defines, creates, dynamically deploys and supports such scripts. SensorWare is designed for iPAQ devices with megabytes of RAM. The

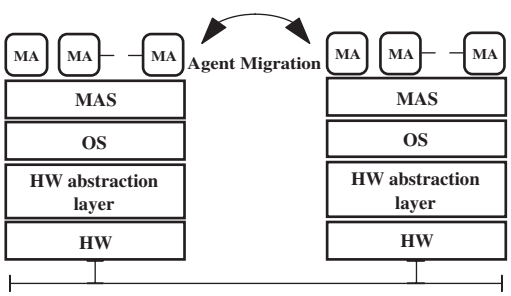


FIGURE 1. A general mobile-agent-oriented sensor node architecture.

verbose program representation and on-node Tcl interpreter can be acceptable overheads; however, they are not yet on a sensor node.

Agilla [5] is an agent-based middleware where each node supports multiple agents and maintains a tuple space and neighbor list. The tuple space is local and shared by the agents residing on the node. Special instructions allow agents to remotely access another node's tuple space. The neighbor list contains the address of all one-hop nodes. Agents can migrate carrying their code and state, but do not carry their own tuple spaces. Agilla is currently implemented on MICA2, MICAZ and TelosB motes.

While both Agilla and SensorWare rely on mobile agents they employ a different communication model: Agilla's agent interaction is based on local tuple spaces, whereas SensorWare's agent interaction is based on direct communication based on network messages. In [13] another mobile agent framework is proposed. The framework is implemented on Crossbow MICA2DOT motes. In particular, it provides agent migration and agent interaction based both on local shared memory and network messages. In [14] the authors propose an extension of Agilla to support direct communication based on messages. In particular, to establish direct communications, agents are mediated by a middle component (named landmark) that interacts with agents through zone-based registration and discovery protocols.

In [6] actorNet, a mobile agent platform for WSNs based on the Actor model is proposed. In particular, it provides services such as virtual memory, context switching and multi-tasking to support a highly expressive yet efficient agent functional language. Currently, the sensor node actorNet platform is specifically designed for TinyOS on Mica2 sensors.

The above described MASs for WSNs [5, 6, 13, 14] are all implemented for TinyOS-based sensor platforms and use *ad hoc* languages for agent programming (Agilla uses a micro-programming language, whereas actorNet employs a functional-oriented language). Although some supported operations (e.g. migration) are very efficient, programming complex tasks is not so straightforward and, moreover, developers need to learn another very specific language.

Finally, the Java-based AFME [15], a lightweight version of the agent factory framework purposely designed for wireless pervasive systems and implemented in J2ME, has been recently ported onto Sun SPOT and purposely used for exemplifying agent communication and migration in WSNs [10]. However, AFME was not specifically designed for WSNs and, particularly, for Java Sun SPOT. MAPS, the Java-based agent platform proposed in this paper, is conversely specifically designed for WSNs and fully uses the release 4.0 (blue) of the Sun SPOT library to provide advanced functionality of communication, migration, sensing/actuation, timing and flash memory storage. Moreover, it allows developers to program agent-based application in Java according to the rules of the MAPS framework, and thus no translator and/or

interpreter need to be developed and no new language has to be learnt.

3. MAPS ARCHITECTURE AND PROGRAMMING MODEL

In this section requirements, architecture (at system and agent level) and programming model of MAPS are described.

3.1. Requirements

The MAPS framework has been appositely defined for resource-constrained sensor nodes; in particular its requirements are the following:

(i) *Lightweight agent server architecture*. The agent server architecture must be lightweight, which implies the avoidance of heavy concurrency models and, therefore, the exploitation of cooperative concurrency to run agents.
(ii) *Lightweight agent architecture*. The agent architecture must also be lightweight so that agents can be efficiently executed and migrated.
(iii) *Minimal core services*. The main core services must be: agent migration, sensing capability access, agent naming, agent communication and timing. The agent migration service allows an agent to be moved from one sensor node to another by retaining the code, data and execution state. The sensing capability access service allows agents to access the sensing capabilities of the sensor node and, more generally, its resources (actuators, input signalers, flash memory). The agent naming service provides a region-based namespace for agent identifiers and agent locations. The agent communication service allows local and remote one-hop/multi-hop message-based communications among agents. The timing service allows agents to set timers for timing their actions.
(iv) *Plug-in-based architecture extensions*. Any other service must be defined in terms of one or more dynamically installable components (or plug-ins) implemented as single mobile agents or cooperating mobile agents.
(v) *Layer-oriented mobile agents*. Mobile agents may be natively characterized on the basis of the layer to which they belong: application, middleware and network layer. They should also be able to locally interact to enable cross-layering.

3.2. Agent server architecture

The designed sensor node architecture is shown in Fig. 2. The architecture is based on components that interact through events. The choice to design the architecture according to a component- and event-based approach is motivated by the effectiveness that such a kind of architecture has demonstrated for sensor node programming. In fact, the TinyOS operating system [16], the *de facto* standard for motes, relies on this kind of architecture. In particular, the main components are the following:

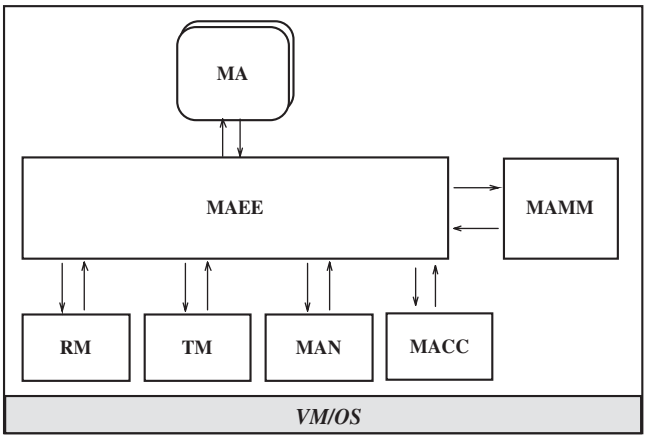


FIGURE 2. The sensor node architecture.

(1) *Mobile agent* (*MA*). The MAs are computing components which are differentiated on the basis of the layer (application, middleware and network) at which they perform tasks. Application layer MAs incorporate application-level logic performing sensor monitoring, actuator control, data filtering/aggregation, high-level event detection, application-level protocols etc. Middleware layer MAs perform middleware-level tasks such as distributed data fusion, discovery protocols for agents, data and sensors, scope management etc. Network layer MAs mainly implement transport (e.g. data dissemination) and network (e.g. multi-hop routing) protocols. Agents at different layers can locally interact to implement cross-layering.
(2) *Mobile agent execution engine* (*MAEE*). The MAEE is the component that supports the execution of agents by means of an event-based scheduler enabling cooperative concurrency. The MAEE handles each event emitted by or to be delivered at MAs through decoupling event queues. The MAEE interacts with the other core components to fulfill service requests (message transmission, sensor reading, timer setting etc.) issued by the MAs.
(3) *Mobile agent migration manager* (*MAMM*). The MAMM component supports the migration of agents

from one sensor node to another. In particular, the MAMM is able to: (i) serialize an MA into a message and send it to the target sensor node and (ii) receive a message containing a serialized MA, deserialize and activate it. The agent serialization format includes the code, data and execution state.

(4) *Mobile agent communication channel* (*MACC*). The MACC component enables inter-agent communications based on asynchronous messages. Messages can be unicast, multicast or broadcast.

(5) *Mobile agent naming* (*MAN*). The MAN component provides agent naming based on proxies and regions [17] to support the MAMM and MACC components in their operations. The MAN also manages the (dynamic) list of the neighbor sensor nodes.

(6) *Timer manager* (*TM*). The TM component provides the timer service that allows for the management of timers to be used for timing MA operations.

(7) *Resource manager* (*RM*). The RM component provides access to the sensor node resources: sensors/actuators, battery and flash memory.

3.3. Agent programming model

The architecture of an MA is modeled as a multi-plane state machine communicating through events (see Fig. 3). This architecture allows exploiting the benefits derived from three paradigms for WSN programming: event-driven programming [16], state-based programming [18] and agent-based programming [5]. Moreover, it enables role-based programming, an important paradigm for agents, as agents behave differently according to the role they can assume during their lifecycle [19].

In particular the architecture consists of:

(i) *Global variables* (*GV*). The GV component represents the data of the MA including the MA identity.

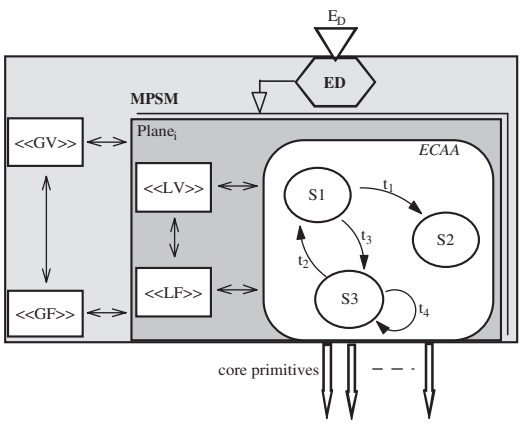


FIGURE 3. The mobile agent architecture.

(ii) *Global functions* (*GF*). The GF component consists of a set of supporting functions which can access GV but can invoke neither core primitives nor other functions.

(iii) *Multi-plane state machine* (*MPSM*). The MPSM component consists of a set of planes. Each plane may represent the behavior of the MA in a specific role. In particular a plane is composed of:

(a) *Local variables* (*LV*). The LV component represents the local data of a plane.

(b) *Local functions* (*LF*). The LF component consists of a set of local plane supporting functions which can access LV but can invoke neither core primitives nor other functions.

(c) *ECA-based automata* (*ECAA*). The ECAA component represents the dynamic behavior of the MA in that plane and is composed of states and mutually exclusive transitions among states. Transitions are labeled by ECA rules: E[C]/A, where E is the event name, [C] is a boolean expression based on the GV and LV variables, and A is the atomic action. A transition t is triggered if t originates from the current state (i.e. the state in which the ECAA component is), the event with the event name E occurs and [C] holds. When the transition fires, A is first executed and, then, the state transition takes place. In particular, the atomic action can use GV, GF, LV and LF for performing computations and, particularly, invoking the core primitives (see Fig. 4) to asynchronously emit one or more events. The delivery of an event is asynchronous and can occur only when the ECAA is idle, i.e. the handling of the last delivered event (ED) is completed.

(iv) *Event dispatcher* (*ED*). The ED component dispatches the event delivered by the MAEE to one or more planes according to the events that the planes are able to handle. In particular, if an event must be dispatched to more than one plane, the event dispatching is appositely serialized.

4. THE MAPS FRAMEWORK

The implementation of MAPS is a real challenge due to the constrained resources of the current sensor nodes. Nevertheless, due to recent advances in operating systems and virtual machines as well as sensor technologies, an actual implementation could be done in nesC/TinyOS on TelosB motes or in Java on Sun SPOT nodes [8]. Although the implementations of the currently available mobile agent frameworks for WSN (see Section 2) have to date been carried out in nesC/TinyOS, by also using the Maté virtual machine [20], we believe that the object-oriented features offered by the Sun SPOT technology could provide more flexibility and extendability as well as easiness of development

```
send(SourceMA, TargetMA, EventName, Params, Local)
SourceMA  = id of the transmitting MA
TargetMA  = id of the MA target |
            id of the Group target |
            ALL for event broadcast to neighbors
EventName = name of the event to be sent
Params    = set of event parameters encoded
            as pairs <attribute, value>
Local     = local (true) or remote (false) scoped event

create(SourceMA, MAId, MAType, Params, NodeLoc)
MAId    = id of the MA to be created
MAType  = type of the MA to be created
Params  = agent creation parameters
NodeLoc = node location of the created agent

clone(SourceMA, MAId, NodeLoc)
MAId    = id of the cloned MA
NodeLoc = node location of the cloned agent

migrate(SourceMA, NodeLoc)
NodeLoc = target location of the MA | ALL neighbors

sense(SourceMA, IdSensor, Params, BackEvent)
IdSensor  = id of the sensor
Params    = parameters for sensor readings
BackEvent = notifying event containing the readings

actuate(SourceMA, IdActuator, Params)
IdActuator = id of the actuator
Params     = parameters for actuator writings

input(SourceMA, BackEvent, Params)
Params     = parameters for switch selection
BackEvent  = event notifying the input captured from the
selected switch(es)

flash(SourceMA, Params, BackEvent)
Params     = flash memory access parameters
BackEvent  = event notifying the completion of the flash
memory operation (if it is a read operation, it contains
the read data)

setTimer(SourceMA, Params, BackEvent)
Params     = timer parameters
BackEvent  = event notifying the timer firing

resetTimer(SourceMA, IdTimer)
IdTimer    = id of the timer to reset
```

FIGURE 4. The prototypal core primitives.

for an efficient implementation of the proposed framework. The Sun SPOT sensor nodes are based on the Squawk VM [9] which is fully Java compliant and CLDC 1.1-compatible. In particular, the offered features are the following:

(i) *Java programming language*. Sensor node software is programmed in the Java language by using Java standard libraries and specific Sun SPOT libraries such as main Sun SPOT board classes, sensor board transducer classes and Squawk operating environment classes.

(ii) *NetBeans IDE for software development*. The IDE fully supports code editing, compilation, deployment and execution for Sun SPOTs. This enables a more rapid software prototyping.

(iii) *Single-hop/multi-hop and reliable/unreliable communications*. The current version of the Sun SPOT SDK uses the GCF (Generic Connection Framework) to provide radio communication between SPOTs, routed via multiple hops if necessary. Two protocols are available: the radiostream protocol and the radiogram protocol. The radiostream protocol provides reliable, buffered, stream-based communication between two devices. The radiogram protocol provides datagram-based communication between two devices and broadcast communications. This protocol provides no guarantees about delivery or ordering. Datagrams sent over more than one hop could be silently lost, be delivered more than once and be delivered out of sequence. Datagrams sent over a single hop will not be silently lost or delivered out of sequence, but they could be delivered more than once. The protocols are implemented on top of the MAC layer of the 802.15.4 implementation.

(iv) *Easy access to the sensor node devices* (*sensors, flash memory, timer, battery*). The Sun SPOT device libraries contains drivers to easily access and use the following: the on-board LED, the PIO, AIC, USART and Timer-Counter devices in the AT91 package, the CC2420 radio chip (in the form of an IEEE 802.15.4 Physical interface), an IEEE 802.15.4 MAC layer, an SPI interface (used for communication with the CC2420 and off-board SPI devices) and an interface to the flash memory.

(v) *Code migration support*. An Isolate is a mechanism by which an application is represented as an object. In Squawk, one or more applications can run in the single JVM. Conceptually, each application is completely isolated from all other applications. The Squawk implementation has the interesting feature of Isolate migration, i.e. an Isolate running on one Squawk VM instance can be paused, serialized to a file or over a network connection and restarted in another Squawk VM instance.

MAPS is implemented on the basis of the aforementioned Java Sun SPOT features which fully provide support to the implementation of each component introduced in Section 3.2. In the following subsections the main MAPS classes (see Fig. 5) and related functionalities are described (more implementation details as well as the MAPS framework code ver. 1.1 can be found in [21]).

The sensor node components are threads that can be instantiated through a Factory class based on the Singleton pattern [22]. Such components are actually created at the node bootstrap when the MobileAgentServer is instantiated by the main application MIDlet. The MobileAgentServer creates the MobileAgentExecutionEngine which, in turn, creates all the other components. As soon as the MobileAgentExecutionEngine starts, it activates an InterIsolateServer to communicate with mobile agent components and broadcasts a *discovery publish* event to announce itself to the neighbor agent-based sensor nodes. After the creation of the MobileAgentServer, mobile agent components can be added to it by the *addAgent* method.

The MobileAgentExecutionEngine is the core component which exposes the interface for supporting all the primitives

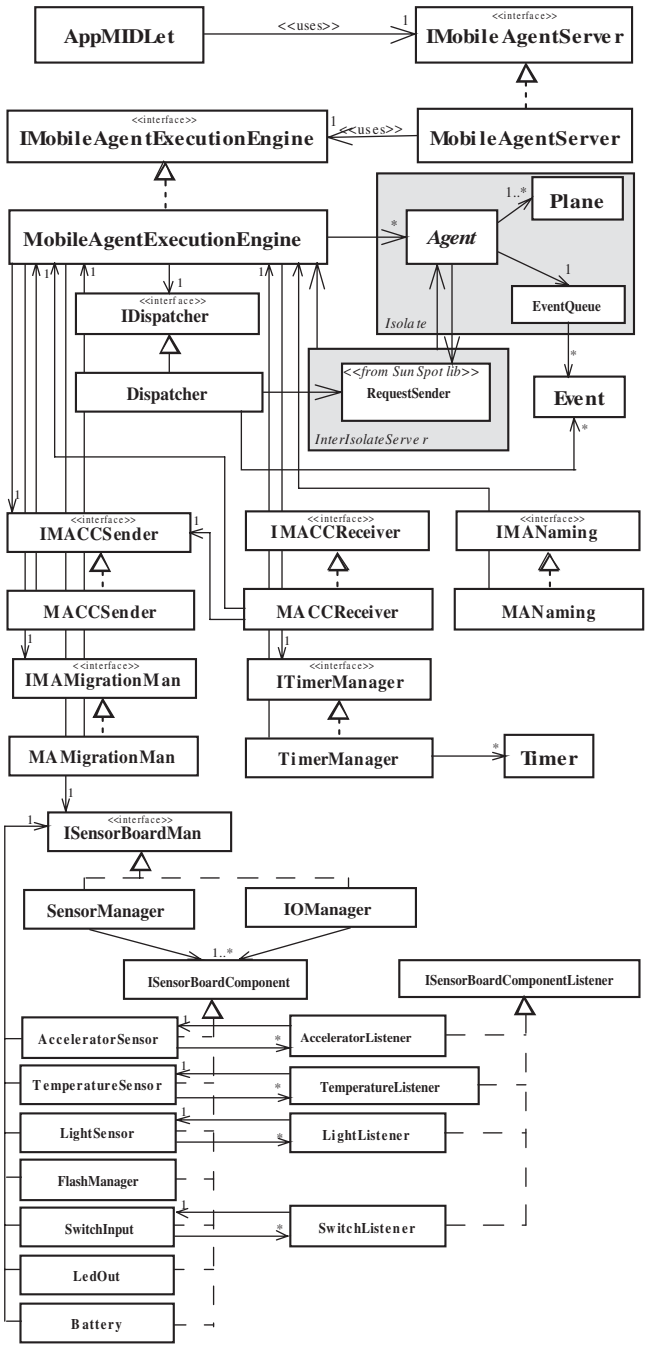


FIGURE 5. A simplified class diagram of the MAPS framework.

defined in Section 3.3 (see Fig. 4). The communication among agents, between agents and system components and, sometimes, among components are based on Event objects. An Event object is composed of:

(i) *sourceID*, which is the agent/component identifier of the event source;
(ii) *targetID*, which is the agent/component identifier of the event target;
(iii) *typeName*, which represents the name of the event types that are grouped according to their specific function/component (see Table 1 for the most important ones);
(iv) *params*, which include the event data organized as a chain of pairs <key, value>;
(v) *durationType*, which specifies the event duration. It can assume the following three values:

(a) NOW, for instantaneous events;
(b) FIRST_OCCURRENCE, for events that wait for the first occurrence of a specific value;
(c) PERMANENT. In this case, the event is sent every time values set in the event parameters are met.

A mobile agent runs in a thread supported Isolate that is instantiated at agent creation time. It is composed of an event queue which contains the Event objects delivered to the agent by the Dispatcher but not yet processed, and the multi-plane state machine containing the dynamic agent behavior. Interaction between mobile agents and the MobileAgentExecutionEngine is made possible by the InterIsolate server and enabled by its RequestSender component (based on the Sun SPOT library).

Remote inter-agent communication is enabled by MACCSender and MACCReceiver component which, respectively, allows transmitting and receiving network messages according to the radiogram protocol.

The MANaming component allows managing the list of neighbor sensor nodes and agents by means of a lightweight beaconing-based announcement protocol based on broadcast messages supported by the radiogram protocol. Moreover, agent proxy components [14] are used to route network messages to migrated mobile agents.

The MAMigrationMan component manages the migration process of a mobile agent from one sensor node to another. To this purpose, it uses the methods provided by the SquawkVM to hibernate/dehibernate and serialize/deserialize an isolate. However, as dynamic class loading is not yet supported by the current version (v4.0 blue) of the Sun SPOT libraries, the agent code should reside at the destination node. In particular, the migration process, which is single-hop and reliable, is implemented as follows: (i) the agent destination node is contacted through a specific message which causes the opening of a socket waiting for an incoming request based on the radiostream protocol; (ii) the agent destination node sends an ACK back to the agent source node; (iii) the source node therefore establishes a radiostream connection with the destination node; (iv) the mobile agent is paused, hibernated, serialized into a byte array and sent over the connection to the destination; and (v) at the destination node, the mobile agent is received, deserialized, dehibernated and reactivated.

TABLE 1. Event types for functions and components.

Function/Component	Types	Usage description
Agent management	AGN_CREATION	Request an agent creation
	AGN_ID	Signal the ID of the created agent
	AGN_START	Start an agent
	AGN_TERMINATED	Terminate an agent
Migration	MGR_EXECUTED	Reactivate a migrated agent
	MGR_REQUEST	Request Migration an agent migration
	MGR_ACK	Signal an accomplished migration
SPOT discovery	DSC_PUBLISH	Publish a node discovery message
	DSC_ANSWER	Signal a discovered node
	DSC_REFRESH	Refresh the neighbor nodes
Message send/receive	MSG	Request a msg transmission
	MSG_TO_BASESTATION	Request a msg transmission to the BS
Timer	TMR_EXPIRED	Signal a timer expiration
Execution Engine	EXE_GET_LOCAL_AGENTS	Request the IDs of the local agents
	EXE_GET_NEIGHBORS	Request the IDs of the neighbor agents
Led	LED_ON	Request a led to be turned on
	LED_OFF	Request a led to be turned off
	LED_BLINK	Request a led to blink
Switch	SWT_PRESSED_RELEASED	Prepare a reading from a switch
Temperature sensor	TMP_CURRENT	Request the current temp value
Light sensor	LGH_CURRENT	Request the current light value
Accelerometer sensor	ACC_CURRENT	Request the current acceleration value
	ACC_TILT	Request the current tilt value
Flash	FLS_ADD	Request to write byte to the flash
	FLS_GET	Request to read byte from the flash
Battery	BTR_CURRENT_LEVEL	Request the current battery level

The TimerManager component handles Timer objects which can be requested by mobile agents to time their operations. Timers can be one-shot or periodic.

Finally the SensorManager component manages available sensors (accelerometer, light and temperature) and actuators (e.g. LEDs), whereas the IOManager component manages input from switches and the flash memory.

4.1. A programming example: mobile agent-based remote sensor monitoring

The example agent-based application for monitoring remote sensors is structured in three agents:

(i) DataCollectorAgent, which collects data sensed from the temperature, light and accelerometer sensors, and the battery of the Sun SPOT node;
(ii) DataMessengerAgent, which carries collected sensed data from the sensing node to the base station;
(iii) DataViewerAgent, which displays the received collected data.

After application deployment and execution, the DataViewerAgent sends a message to the DataCollectorAgent to start its activity as soon as the user presses a switch on the Sun SPOT on which the DataViewerAgent is running. The DataCollectorAgent therefore begins its collecting activity and as soon as the user pushes a switch on the Sun SPOT on which the DataCollectorAgent is running, it creates the DataMessengerAgent with the collected data that migrates to the DataViewerAgent node for data visualization. Finally, the monitoring activity terminates when the user presses again a switch of the Sun SPOT on which the DataViewerAgent is running. The sequence of interactions among the three defined agents is shown in Fig. 6 through an M-UML sequence diagram [23]. This simple yet effective application, deployed on just two sensor nodes, allows for testing the most important mechanisms provided by MAPS.

For the sake of illustrating MAPS-based programming, the state machine of the DataCollectorAgent plane along with the action code, which uses the MAPS library, is shown in Fig. 7 and briefly explained in the following. The AGN_Start event causes the transition from the creation state to the STARTTIMER state and the execution of an example operation on the flash memory (action A0), i.e. adding some data to the flash space of the agent. In the STARTTIMER state, when the network message (MSG) sent by the DataViewerAgent arrives and the guard [go == true]

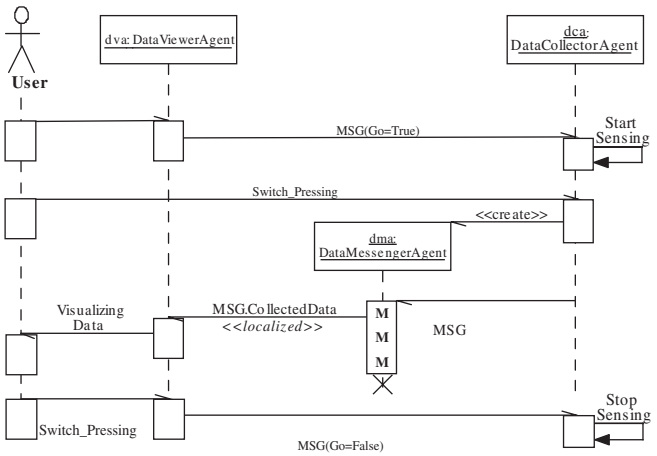


FIGURE 6. The M-UML sequence diagram of the interactions among the agents.

holds, a timer timing the sensing operations is set up to fire after 3 s, some actuation on the LEDs is requested and some input from the switches is ready to be read (action A1). When the timer fires (see TMR_Expired event), the sensing operations are requested (action A2). When such operations are completed (see actions A5–A8), the guard [dataColl = = numData] holds so that data are collected and, as a visual signal, an LED is actuated to blink blue (see action A9). When the switch is pressed by the user, a DataMessengerAgent is created (action A3) and the collected data are passed to it (action A10) when the AGN_Id event, containing the agent ID of the created agent, is received. When the event MSG is received and the guard [go = = false] holds, the agent is terminated (action A4).

5. PERFORMANCE EVALUATION

The used testbed for testing and evaluation consists of a Sun SPOT kit (two sensor nodes and one base station) with the SDK 4.0 version (blue). The MAPS framework has a memory occupation (without any running agent) of about 70 kB in central memory, keeping free a space of 378 kB. Such space can be exploited for agent execution. The agent developed for the agent migration benchmarking (see below) needs 22 kB of central memory. The space occupied by the jar of MAPS on the flash memory is 92 kB out of the 4 MB available [8]. To evaluate the performance of MAPS three micro-kernel benchmarks have been defined according to [24] for the following mechanisms:

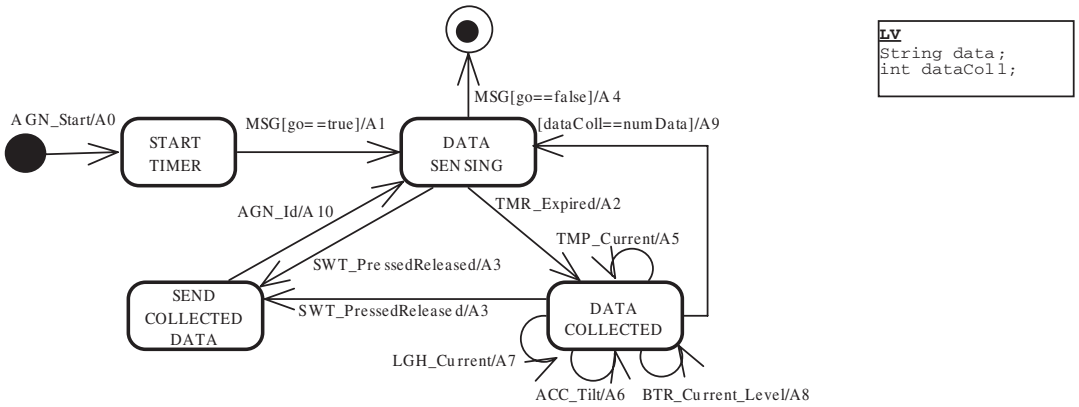
(1) *Agent communication*. The agent communication time is computed for two agents running onto different nodes and communicating in a client/server fashion (request/reply). Two different request/reply schemes are used: (i) *Data B&F*, in which both request and reply contain the same amount of data and (ii) *Data B*, in which only the reply contains data. Results are shown in Fig. 8. By increasing the amount of data, communication times linearly increase. The noticed overhead of MAPS communication with respect to the cases without MAPS is mainly due to the message format of MAPS which contains event parameters (see Section 4).

(2) *Agent creation*. The agent creation time is computed for agents having different number of planes ranging from 1 to 51. Figure 9 reports the results which show that the creation time is linear with respect to the number of planes.

(3) *Agent migration*. The agent migration time is calculated for an agent ping-pong among two single-hop-distant sensor nodes. It has been computed in the following two cases: (i) *with MAPS*, which uses the complete functionality of MAPS and (ii) *without MAPS*, which does not use the MAPS engine and migration manager but just the Java SunSPOT library. This allows highlighting the overhead introduced by the framework for having complete migration reliability. Migration times are computed by varying the data cargo of the ping-pong agent. Although migration performances without MAPS are better, complete reliability of agent migration is not guaranteed. The obtained migration times (see Fig. 10) are high due to the slowness of the SquawkVM operations supporting the migration process. In particular, serialization is a very costly operation: serialization of the ping-pong agent on an average takes 4.5 s. Moreover, radiostream connections are very slow to guarantee reliability.

6. A CASE STUDY: REAL-TIME HUMAN ACTIVITY MONITORING

The effectiveness of MAPS to support WBSNs applications [25] is demonstrated by the development of a real-time activity recognition system prototype. Nowadays WBSNs have great potential to enable a broad variety of assisted living applications such as human biophysical/biochemical control for health care, human activity monitoring for health care, e-fitness and emergency detection, and emotional recognition for social networking, security and highly interactive games. It is therefore important to provide design methodologies and programming frameworks that enable rapid prototyping of WBSN applications. Several effective application development frameworks have been already proposed for WBSNs based on TinyOS-based sensor platforms, for example CodeBlue, signal processing in node environment (SPINE) and Titan. In particular, SPINE [26] is a domain-specific framework for collaborative WBSNs which provides effective APIs (libraries of protocols, utilities and data processing functions) and tools (remote configuration of sensors, data gathering and visualization) for signal processing-based applications for the analysis and the classification of sensor data. This provides application developers with an abstraction that improves



Actions

```
A0: byte [] fls = new byte[]{12,13,14,15,16};
    Event l = new Event(agent.getId(), agent.getId(), Event.FLS_ADD, Event.NOW);
    agent.flash(l, fls);
A1: Event timer =new Event(agent.getId(), agent.getId(), Event.TMR_EXPIRED, Event.NOW );
    timerID = agent.setTimer(true, 3000, timer);
    Event blink = new Event(agent.getId(), agent.getId(), Event.LED_BLINK, Event.NOW);
    blink.setParam(ParamsLabel.LED_INDEX, "0");
    blink.setParam(ParamsLabel.LED_COLOR, "blue");
    agent.actuate(blink);
    Event swtPressed = new Event(agent.getId(),agent.getId(),Event.SWT_PRESSED_RELEASED,Event.PERMANENT);
    swtPressed.setParam(ParamsLabel.SWT_PRESSED, "false");
    swtPressed.setParam(ParamsLabel.SWT_RELEASED, "true");
    swtPressed.setParam(ParamsLabel.SWT_INDEX, "2");
    agent.input(swtPressed);
A2: Event temp = new Event(agent.getId(), agent.getId(), Event.TMP_CURRENT, Event.NOW);
    temp.setParam(ParamsLabel.TMP_CELSIUS, "true");
    agent.sense(temp);
    Event accel = new Event(agent.getId(), agent.getId(), Event.ACC_TILT, Event.NOW);
    agent.sense(accel);
    Event light = new Event(agent.getId(), agent.getId(), Event.LGH_CURRENT, Event.NOW);
    agent.sense(light);
    Event battery = new Event(agent.getId(), agent.getId(), Event.BTR_CURRENT_LEVEL, Event.NOW);
    agent.sense(battery);
A3: agent.create("test.Messenger", null, agent.getMyIEEEAddress().asDottedHex());
A4: this.terminateAgent();
A5: data+=event.getParam(ParamsLabel.TMP_TEMPERATURE_VALUE) + "-";
    dataColl++;
A6: data+=event.getParam(ParamsLabel.ACC_TILT_X_VALUE) + "-";
    dataColl++;
A7: data+=event.getParam(ParamsLabel.LGH_LIGHT_VALUE) + "-";
    dataColl++;
A8: data+=event.getParam(ParamsLabel.BTR_BATTERY_VALUE) + "-";
    dataColl++;
A9: data+="|";
    Event blink = new Event(agent.getId(), agent.getId(), Event.LED_BLINK, Event.NOW);
    blink.setParam(ParamsLabel.LED_INDEX, "0");
    blink.setParam(ParamsLabel.LED_COLOR, "blue");
    agent.actuate(blink);
    dataColl = 0;
A10:Event msg = new Event(agent.getId(), messengerAgentID, Event.MSG, Event.NOW);
    msg.setParam("collectedData", data);
    agent.send(agent.getId(), messengerAgentID, msg, true);
    data = "";
```

FIGURE 7. The DataCollectorAgent behavior composed of one plane.

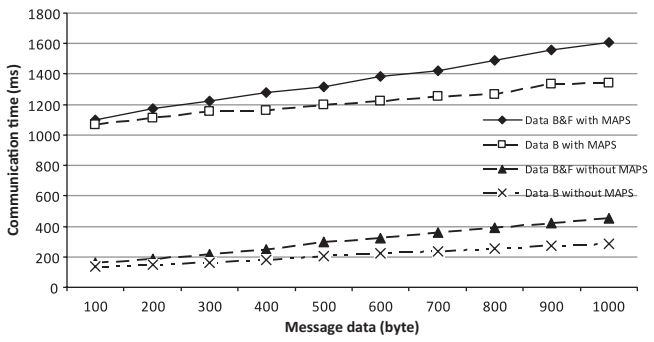


FIGURE 8. Agent communication: request/reply time.

interoperability and allows reducing application development time. SPINE is currently developed in the TinyOS and Z-Stack environments (node side) [27] and in Java (coordinator side).

In this paper, the developed MAPS-based prototype, which relies on SPINE at the base station side, aims at recognizing human postures (e.g. lying down, sitting or standing still) and movements (e.g. walking). The architecture of the system, shown in Fig. 11, is a typical star-based Body Sensor Network (BSN) composed of a base station and two sensor nodes.

On the base station the Java-based SPINE coordinator [26], developed in the context of the SPINE project [28], is resident. The SPINE Manager is used by end-user applications

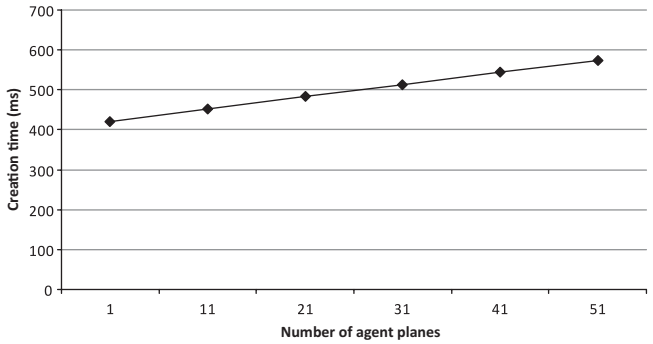


FIGURE 9. Agent creation time.

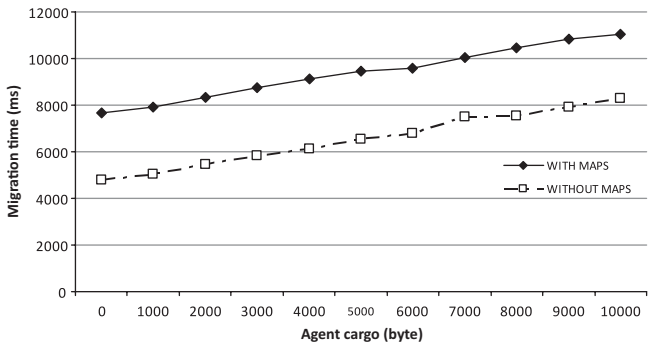


FIGURE 10. Agent migration: ping-pong time.

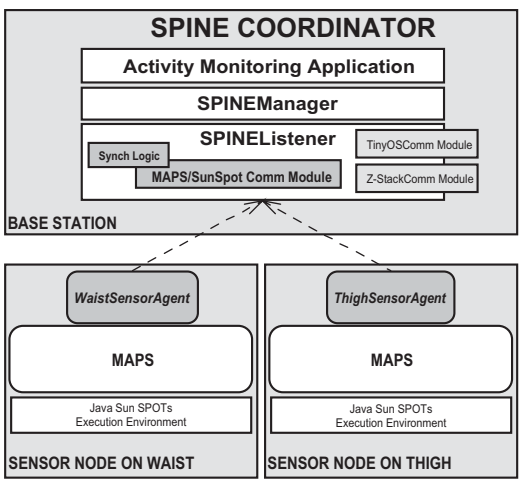


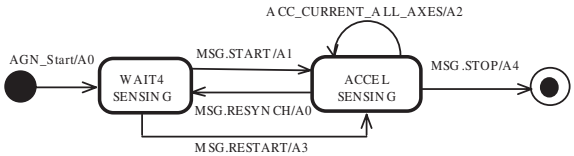
FIGURE 11. Architecture of the agent-based real-time activity recognition system.

(e.g. activity monitoring application) for issuing commands to the BSN. Moreover, the SPINE Manager is responsible of capturing low-level messages and node events through the SPINE Listener to notify registered applications with higher-level events and messages content. A SPINE Comm Module (currently implemented for TinyOS and Z-Stack) is internally composed of a send/receive interface and some components that implement such interface according to the specific sensor platform and formalize the high-level SPINE messages in sensor platform-specific messages. In this work, The SPINE coordinator has been enhanced with a new MAPS/SunSpot communication module (named SunSPOTWSNConnection) to configure and communicate with MAPS-based sensor nodes. Such module translates high-level SPINE messages formatted according to the SPINE OTA (over-the-air) protocol [28] into lower-level MAPS/SunSPOT messages (named SunSPOTMessage) through its transmitter component and vice versa through its receiver component (named SunSPOTReceiver). The SunSPOTReceiver also integrates an application-specific logic for the synchronization of the two sensors (see below). The activity monitoring application as well as the SPINE Manager was thus completely reused; only the SPINE Listener was modified by such an enhancement.

The sensor nodes based on the Java Sun SPOTs are, respectively, located on the waist and on the thigh of the monitored person. In particular, MAPS is deployed on the sensor nodes and supports the execution of the WaistSensorAgent and the ThighSensorAgent. The WaistSensorAgent and the ThighSensorAgent have similar behavior: (i) sensing the three axial accelerometer sensors according to a given sampling time (ST = 1/sampling_rate); (ii) computation of specific features (Mean, Max and Min functions) on the acquired raw data according to the window (W) and shift parameters (S), i.e. W is the sample size on which features are computed, whereas S is the percentage of sliding on W; and (iii) features aggregation and transmission to the coordinator. While the values of the W and S parameters are equally set for both agents, the agents differ in the specific computed features: the WaistSensorAgent computes the mean values for data sensed on the XYZ axes, the min and max values for data sensed on the X-axis, whereas the ThighSensorAgent calculates the min value for data sensed on the X-axis. The behavior of the sensor agents is specified through two planes: the *sensing* plane and the *feature calculation and transmission* plane. In Fig. 12 the behavior of the WaistSensorAgent is reported (the behavior of the ThighSensorAgent complies with the same structure but the computed features are different as discussed above).

With reference to Fig. 12a, after an initialization action (A0) driven by the occurrence of the AGN_START event, the *sensing* plane goes into the WAIT4SENSING state. The MSG.START event allows starting the sensing process by the execution of action A1: (i) sensing parameters (W, S, ST), data acquisition buffers for XYZ channels of the accelerometer sensor (windowX, windowY, windowZ) and data buffers for feature calculation (windowFE4X, window FE4Y, window FE4Z) are initialized (see *initSensingParamsAndBuffers* function); (ii) the timer is set for timing the data acquisition according to the ST parameter (see *timerSetForSensing* function; in particular the highly precise Sun SPOT timer is used instead of the timer provided by MAPS as in the example of Section 4.1);

(a)




```
GV
byte timestamp;
double [] windowX4FE,
double [] windowY4FE,
double [] windowZ4FE;
byte bodyPosition;
String basestationAddress;
LV
int W;
int S;
int ST;
double [] windowX,
double [] windowY,
double [] windowZ;
byte sampleCounter;
int nextSampleIndex;
IAT91_TC timer;
```

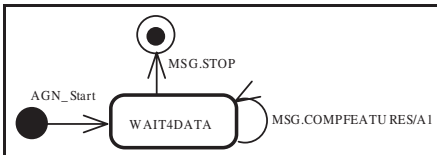
Actions

```
A0: initVars();
A1: initSensingParamsAndBuffers(event);
    timerSetForSensing();
    doSensing();
A2: bufferFilling(event);
    sampleCounter++;
    nextSampleIndex=(nextSampleIndex+1)%W;
    if (sampleCounter==S){
     sampleCounter=0;
     copySensingBuffersIntoBuffersForComputingFeatures();
     Event msgToItself = new Event (agent.getId(), agent.getId(), Event.COMPFEATURES, Event.NOW);
     agent.send(agent.getId(), agent.getId(), msgToItself, true);
    }
    timerReset();
    doSensing();
A3: timerDisabling();
    initVars();
    A1;
A4: timerDisabling();
```

LF

```
initVars():
  sampleCounter=0;
  nextSampleIndex=0;
  agent.timestamp=0;
initSensingParamsAndBuffers(Event event):
  (WaistSensorAgent)agent.bodyPosition=Byte.parseByte(event.getParam("BODY_POSITION"));
  (WaistSensorAgent)agent.basestationAddress=event.getParam("BASESTATION_ADDRESS");
  W=Integer.parseInt(event.getParam("WINDOW_SIZE"));
  S=Integer.parseInt(event.getParam("SHIFT_SIZE"));
  ST=Integer.parseInt(event.getParam("SAMPLE_RATE_MS"));
  windowX = new double[W]; windowY = new double[W]; windowZ = new double[W];
  (WaistSensorAgent)agent.windowX4FE = new double[W]; (WaistSensorAgent)agent.windowY4FE = new double[W];
  (WaistSensorAgent)agent.windowZ4FE = new double[W];
timerSetForSensing():
  timer = Spot.getInstance().getAT91_TC(0);
  int cnt = (int)(ST * 1000 / 2.1368);
  timer.configure(TimerCounterBits.TC_CAPT | TimerCounterBits.TC_CPCTRG | TimerCounterBits.TC_CLKS_MCK128);
  timer.setRegC(cnt);
  timer.enableAndReset();
  timerReset();
doSensing():
  Event accel = new Event(agent.getId(), agent.getId(), Event.ACC_CURRENT_ALL_AXES, Event.NOW);
  agent.sense(accel);
bufferFilling(Event event):
  windowX[nextSampleIndex] = Double.parseDouble(event.getParam(ParamsLabel.ACC_ACCEL_X_VALUE));
  windowY[nextSampleIndex] = Double.parseDouble(event.getParam(ParamsLabel.ACC_ACCEL_Y_VALUE));
  windowZ[nextSampleIndex] = Double.parseDouble(event.getParam(ParamsLabel.ACC_ACCEL_Z_VALUE));
timerReset():
  timer.enableIrq(TimerCounterBits.TC_CPCS);
  timer.waitForIrq();
  timer.status();
timerDisabling():
  timer.disable();
  timer.shutDown();
```

(b)




```
GV
byte timestamp;
double [] windowX4FE,
double [] windowY4FE,
double [] windowZ4FE;
String basestationAddress;
LV
double [] resultsX;
double [] resultsY;
double [] resultsZ;
```

Actions

```
A1: Event msgToServer = new Event  (this.agent.getId(), Constants.MSG_TO_BASESTATION,
                                    Event.MSG_TO_BASESTATION, Event.NOW);
    msgToServer.setParam(ParamsLabel.AGT_BASESTATION_ADDRESS, (WaistSensorAgent)agent.basestationAddress);
    resultsX = meanMaxMin((WaistSensorAgent)agent.windowX4FE);
    resultsY = meanMaxMin((WaistSensorAgent)agent.windowY4FE);
    resultsZ = meanMaxMin((WaistSensorAgent)agent.windowZ4FE);
    msgToServer.setParam("BodyPosition", "Waist");
    msgToServer.setParam("MeanX","" + resultsX[0]);
    msgToServer.setParam("MeanY","" + resultsY[0]);
    msgToServer.setParam("MeanZ","" + resultsZ[0]);
    msgToServer.setParam("MaxY", "" + resultsY[1]);
    msgToServer.setParam("MinY", "" + resultsX[2]);
    (WaistSensorAgent)agent.timestamp=((WaistSensorAgent)agent.timestamp+1)%128;
    msgToServer.setParam("Timestamp", "" + (WaistSensorAgent)agent.timestamp);
    agent.send(agent.getId(), Constants.MSG_TO_BASESTATION, msgToServer, false);
```

LF

```
double [] meanMaxMin(double []): //omissis
```

FIGURE 12. Two-plane behavior of the WaistSensorAgent: (a) sensing plane and (b) feature calculation and transmission plane.

and (iii) a data acquisition is requested by submitting the ACC_CURRENT_ALL_AXES event by the *sense* primitive (see *doSensing* function). Once the data sample is acquired, the ACC_CURRENT_ALL_AXES event is sent back with the acquired data and the action A2 is executed: (i) the buffers are circularly filled with the proper values (see *bufferFilling* function); (ii) the sampleCounter is incremented and the nextSampleIndex is incremented module W for the next data acquisition; (iii) if S samples have been acquired, features are to be calculated; thus sampleCounter is reset, samples in the buffers are copied into the buffers for computing features and the COMPFEATURES event is sent to itself for being processed by the other plane; (iv) the timer is reset; and (v) data acquisition is finally requested. In the ACCELSENSING state the MSG.RESYNCH might be received for resynchronization purposes (see below); it brings the *sensing* plane into the WAIT4SENSING state. The MSG.RESTART brings the *sensing* plane back into the ACCELSENSING state for (reconfiguring and) continuing the sensing process. The MSG.STOP eventually terminates the sensing process.

The *feature calculation and transmission* plane (see Fig. 12b) is much simpler than the *sensing* plane. After it is started, in the WAIT4DATA it waits for the reception of the MSG.COMPFEATURES state which will be sent by the *sensing* plane once S samples have been acquired (see above). Such an event triggers the calculation of the features through the *meanMaxMin* function and their transmission to the base station by sending the MSG_TO_BASESTATION event appropriately constructed.

An important issue is the synchronization between the operations of the two agents which is to be maintained within a maximum skew for not affecting the real-time monitoring: if such a skew is overtaken, the two agents are to be resynchronized. As the sensor agents compute a different number of features, when the sampling rate is high, the agent computing more features (i.e. the WaistSensorAgent) takes more time to complete its operations for each S sample acquisitions than the ThighSensorAgent. Resynchronization is driven by the Synch Module (see Fig. 11), included in the developed MAPS/SunSpot comm module, which sends a resynchronization message (see Fig. 12a) as soon as it detects that the synchronization skew is greater than a given threshold. Detection is based on the skew time between the receptions of two messages sent by the agents that contain features referring to the same interval of S sample acquisitions: if skew $\geq P * S *$ ST, where P is a percentage and $S = W/2$.

Figure 13 shows the results of some experiments aimed at evaluating the synchronization of the sensor agents and their monitoring continuity. In particular, the experiments are carried out by fixing ST (ms) $= [25, 50, 100]$, $W = [40, 20]$ and $P = [5\%, 10\%, 25\%]$. The defined measurements are: (i) the *packet pair average time* (*PPAT*), which is the average reception time between two consecutive pairs of synchronized packets (same logical timestamp; see the timestamp variable in Fig. 12) containing the computed features (see MSG_TO_BASESTATION event in Fig. 12b) sent by the sensor agents and (ii) the *synch packet percentage* (*SPP*), which is the percentage of resynchronization packets (see RESYNCH event in Fig. 12a) that are sent by the coordinator for resynchronizing the sensor agents, calculated with respect to the total number of received feature packets. The PPAT should be ideally equal to ST $*$ S, that is, the packet pair arrives during each monitoring period and so there is no desynchronization in the average, whereas the SPP should be as much as possible close to 0, that is, a few or no resynchronizations are carried out and thus the monitoring can be continuous as a resynch operation usually takes 600 ms.

As can be seen from Fig. 13, the system cannot support an ST $= 25$ ms because the PPAT is always greater than the ideal value and the SPP is too high. This leads to non-continuous monitoring due to the very frequent resynchronization (SPP $\geq$ 25%). An ST $= 50$ ms can be supported for $P = 25\%$ as the SPP is almost 8%, thus slightly impacting upon the monitoring continuity. The best results are obtained with ST $= 100$ ms and $P = 25\%$ or 15%; they guarantee monitoring continuity due to an SPP $\approx 0\%$ and regularity as the PPAT ≈ 1000 for $S = 10$ and the PPAT ≈ 2000 for $S = 20$. If $P = 5\%$, also an ST $= 100$ ms is not a good value because a skew of $5\% * S *$ ST frequently occurs. It is worth noting that even though a lower ST allows more accurate monitoring, the considered human activities can be well captured by an ST $= 100$ ms, as demonstrated by the experimental results obtained from the real-time human activity monitoring carried out (see below).

According to such considerations the parameters used by the activity monitoring application during the training and the real-time operating phases were fixed as follows: $ST = 100$ ms, $W = 20$ ($S = 10$), $P = 25\%$. In particular, the activity monitoring application relies on a classifier that recognizes postures and movements defined in a training phase. The application integrates two different classifiers: one based on the K-Nearest Neighbor algorithm [29] and the other based on J48 Decision Tree [30]. They were set up through a training phase and tested considering the parameter setting for data acquisition reported above. According to this setting, the features (Min, Max and Mean) are computed on 20 sampled data every new 10 samples acquired by the sensors. In Table 2 the obtained classification accuracy results are reported. The obtained results are good and encouraging if compared with other works in the literature, which use more than two sensors on the human body [31]. Finally, the interested reader can refer to [26] for viewing the snapshots related to the panels (Live Monitor panel visualizing the recognized human activity, Statistics panel reporting statistics about the human activity, Advanced panel for configuring the sensors and the sensing process) of the reference activity monitoring application.

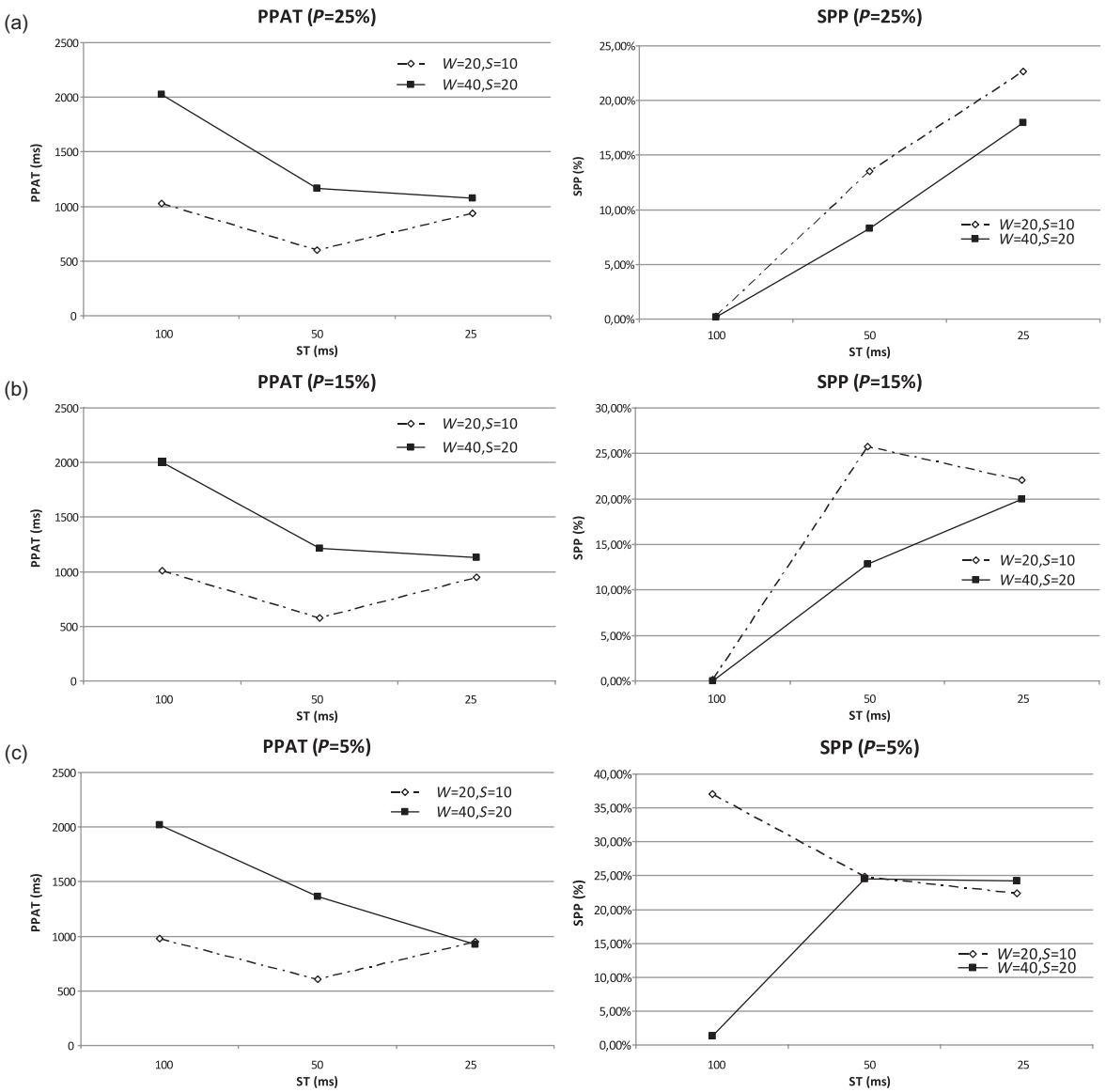


FIGURE 13. Analysis of the synchronization of the sensor agents: PPAT and SPP for (a) $P = 25\%$, (b) $P = 15\%$ and (c) $P = 5\%$.

TABLE 2. Classification accuracy for classifiers based on K-Nearest Neighbor and J48 Decision Tree.

	Walking	Sitting	Standing still	Lying down
K-NN (%)	94	96	92	98
J48 D Tree (%)	92	98	94	94

7. CONCLUSIONS

Programming WSN applications is a complex task that requires suitable programming paradigms and frameworks to cope with the WSN-specific characteristics. Several kinds of micro- and macro-programming techniques have to date been proposed. Among them mobile agent-based programming, which has been formerly introduced for conventional distributed systems, can be more effectively exploited in the context of WSNs. In this paper we have therefore proposed mobile agents as an effective paradigm to program WSN applications and, in particular, presented MAPS, a Java-based framework for the development of agent-based applications for Sun SPOT sensor platforms. By using MAPS, a WSN application can be structured as a set of stationary and mobile agents distributed on sensor nodes supported by a component-based agent execution engine that provides basic services such as message transmission, agent creation, agent cloning, agent migration, timer handling and easy access to the sensor node resources. MAPS programming has been exemplified through a simple

yet effective example that shows how to program the dynamic behavior of agents in terms of state machines on the basis of the MAPS library. Moreover, a complete case study concerning the development and testing of a real-time human activity monitoring system based on WBSNs has been described. It is emblematic of the effectiveness and suitability of MAPS to deal with the programming of complex applications. Finally, we have presented an evaluation of MAPS according to micro-kernel benchmarks (agent communication, migration and creation) usually employed for MASs. Evaluation shows some performance penalties mainly due to very time-consuming operations (Isolate hibernation/serialization and radiostream-based communications) provided by the Sun SPOT libraries and SquawkVM on which MAPS relies.

Ongoing research efforts are being devoted to: (i) further optimizing the communication and migration mechanisms of MAPS; (ii) introducing the fall detection, which allows raising an alarm on detecting that the monitored person has fallen, in the developed real-time human activity monitoring agent-based application; (iii) porting MAPS onto the Sentilla JCreate pervasive computers which are compliant to Java ME CLDC 1.1; (iv) developing an agent-based version of SPINE (named ASpine) through MAPS (a preliminary design of ASpine if reported here [32]).

FUNDING

This work has been partially carried out in the framework of CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

ACKNOWLEDGEMENTS

The authors wish to thank the members of the SPINE project for their useful comments and feedback about the (re)design and implementation of the SPINE-based activity monitoring application through MAPS, and Alessio Carbone for his helpful support in carrying out the performance evaluation of MAPS.

REFERENCES

[1] UCAM-CL-TR-646 (2005) *A survey of Wireless Sensor Network technologies: research trends and middleware's role*. University of Cambridge, UK.

[2] Chen, M., Gonzalez, S. and Leung, V.C.M. (2007) Applications and design issues for mobile agents in wireless sensor networks. *IEEE Wirel. Commun.*, **14**, 20–26.

[3] Silva, A.R., Romao, A., Deugo, A. and Mira da Silva, M. (2001) Towards a reference model for surveying mobile agent systems. *Auton. Agent Multi-Agent Syst.*, **4**, 187–231.

[4] Luck, M., McBurney, P. and Preist, C. (2004) A manifesto for agent technology: towards next generation computing. *Auton. Agents Multi-Agent Syst.*, **9**, 203–252.

[5] Fok, C.-L., Roman, G.-C. and Lu, C. (2005) Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. *Proc. 24th Int. Conf. Distributed Computing Systems* (*ICDCS'05*), Columbus, OH, USA, June 6–10, pp. 653–662. IEEE Computer Society, Washington, DC, USA.

[6] Kwon, Y., Sundresh, S., Mechitov, K. and Agha, G. (2006) ActorNet: An Actor Platform for Wireless Sensor Networks. *Proc. 5th Int. Joint Conf. Autonomous Agents and Multiagent Systems* (*AAMAS*), Hakodate, Japan, May 8–12, pp. 1297–1300. ACM, New York, NY, USA.

[7] Boulis, A., Han, C.-C. and Srivastava, M.B. (2003) Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. *Proc. 1st Int. Conf. Mobile Systems, Applications and Services* (*MobiSys*), San Francisco, CA, USA, May 5–8, pp. 187–200. ACM, New York, NY, USA.

[8] *Sun™ Small programmable object technology* (*Sun SPOT*). (2010) http://www.sunspotworld.com/.

[9] Simon, D. and Cifuentes, C. (2005) The Squawk Java Virtual Machine: Java on the Bare Metal. *Proc. 20th Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA 2005*), San Diego, CA, USA, October 16–20, pp. 150–151. ACM, New York, NY, USA.

[10] Muldoon, C., O'Hare, G.M.P., O'Grady, M.J. and Tynan, R. (2008) Agent Migration and Communication in WSNs. *Proc. 1st Int. Workshop on Sensor Networks and Ambient Intelligence*, held in conjunction with the *9th Int. Conf. Parallel and Distributed Computing, Applications and Technologies*, Dunedin, New Zealand, December 1–4, pp. 425–430. IEEE Computer Society Press, Washington, DC, USA.

[11] *The Sentilla labs*. (2010) http://labs.sentilla.com/.

[12] Aiello, F., Fortino, G. and Guerrieri, A. (2008) Using Mobile Agents as an Effective Technology for Wireless Sensor Networks. *Proc. 2nd IEEE/IARIA Int. Conf. Sensor Technologies and Applications* (*SENSORCOMM 2008*), Cap Esterel, France, August 25–31, pp. 549–554. IEEE Computer Society, Washington, DC, USA.

[13] Szumel, L., LeBrun, J. and Owens, J.D. (2005) Towards a Mobile Agent Framework for Sensor Networks. *Proc. 2nd IEEE Workshop on Embedded Networked Sensors* (*EmNetS-TT*), Sydney, Australia, May 30–31, pp. 79–87. IEEE Computer Society, Washington, DC, USA.

[14] Suenaga, S. and Honiden, S. (2007) Enabling Direct Communication Between Mobile Agents in Wireless Sensor Networks. *Proc. 1st Int. Workshop on Agent Technology for Sensor Networks* (*ATSN-07*), jointly held with the *6th Int. Joint Conf. Autonomous Agents and Multiagent Systems* (*AAMAS-07*), Honolulu, HI, May 14.

[15] *Agent Factory Micro Edition* (*AFME*). (2010) http://sourceforge.net/projects/agentfactory/files/.

[16] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E. and Culler, D. (2003) The nesC Language: A Holistic Approach to Networked Embedded Systems. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (*PLDI 2003*), San Diego, CA, USA, June 9–11, pp. 1–11. ACM, New York, NY, USA.

[17] Welsh, M. and Mainland, G. (2004) Programming Sensor Networks Using Abstract Regions. *Proc. 1st USENIX/ACM Symp. Networked Systems Design and Implementation* (*NSDI'04*),

San Francisco, CA, USA, March 29–31, pp. 3–16. USENIX Association Berkeley, CA, USA.

[18] Kasten, O. and Römer, K. (2005) Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. *Proc. 4th Int. Symp. Information Processing in Sensor Networks*, Los Angeles, CA, USA, April 24–27. IEEE Press, Piscataway, NJ, USA.

[19] Zhu H. and Alkins, R. (2006) Towards Role-Based Programming. *Proc. CSCW'06*, Banff, AB, Canada, November 4–8.

[20] Levis, P. and Culler, D. (2002) Maté: A Tiny Virtual Machine for Sensor Networks. *Proc. 10th Int. Conf. Architectural Support for Programming Languages and Operating Systems* (*ASPLOS X*), San Jose, CA, USA, October 5–9, pp. 85–95. ACM, New York, NY, USA.

[21] *Mobile Agent Platform for Sun SPOT* (*MAPS*). (2010) http://maps.deis.unical.it.

[22] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

[23] Saleh, K., El-Morr, C. and M-UML (2004) An extension to UML for the modeling of mobile agent-based software systems. *Inform. Softw. Technol.*, **46**, 219–227.

[24] Dikaiakos, M., Kyriakou, M. and Samaras, G. (2001) Performance Evaluation of Mobile-Agent Middleware: A Hierarchical Approach. *Proc. 5th IEEE Int. Conf. Mobile Agents*, Atlanta, GA, USA, December 2–4, pp. 244–259. Lecture Notes in Computer Science 2240. Springer, Berlin.

[25] Yang, G.-Z. (2006) *Body Sensor Networks*. Springer.

[26] Gravina, R., Guerrieri, A., Fortino, A., Bellifemine, F., Giannantonio, R. and Sgroi, M. (2008) Development of Body Sensor Network Applications using SPINE. *Proc. IEEE Int. Conf. Systems, Man, and Cybernetics* (*SMC 2008*), Singapore, October 12–15.

[27] Bellifemine, F., Fortino, G., Giannantonio, R. and Guerrieri, A. (2009) Platform-Independent Development of Collaborative WBSN Applications: SPINE2. *Proc. IEEE Int. Conf. Systems, Man, and Cybernetics* (*SMC 2009*), San Antonio, TX, USA, October 11–14.

[28] *Signal Processing In Node Environment* (*SPINE*). (2010) http://spine.tilab.com.

[29] Cover, T. and Hart, P. (1967) Nearest neighbor pattern classification. *IEEE Trans. Inform. Theory*, **13**, 21–27.

[30] Quinlan, R. (1993) *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Francisco, CA, USA.

[31] Maurer, U., Smailagic, A., Siewiorek, D.P. and Deisher, M. (2006) Activity Recognition and Monitoring Using Multiple Sensors on Different Body Positions. *Proc. 3rd Int. Workshop on Wearable and Implantable Body Sensor Networks* (*BSN 2006*), MIT, Boston, MA, USA, April 3–5, pp. 113–116. IEEE Computer Society, Washington, DC, USA.

[32] Bellifemine, F. and Fortino, G. (2009) ASPINE: An Agent-Oriented Design of SPINE. *Proc. Workshop on Objects and Agents* (*WOA'09*), Parma, Italy, July 9–10.