Programming Wireless Body Sensor Network Applications through Agents

Giancarlo Fortino, Stefano Galzarano
Department of Electronics, Informatics and Systems (DEIS)
University of Calabria (UNICAL)
Rende (CS), ITALY
fortino@unical.it, galzarano@si.deis.unical.it

***Abstract*—Wireless Sensor Networks (WSNs) are currently emerging as one of the most disruptive technologies enabling and supporting next generation ubiquitous and pervasive computing scenarios. In particular, Wireless Body Sensor Networks (WBSNs) are conveying notable attention as their real-world applications aim at improving the quality of human beings life by enabling continuous and real-time non-invasive assistance at low cost. This paper proposes a high-level programming approach based on the agent-oriented model to flexibly design and efficiently implement WBSNs applications. The approach is exemplified through a case study concerning a real-time human activity monitoring system which is developed through two different agent-based frameworks: MAPS (Mobile Agent Platform for Sun SPOT) and AFME (Agent Factory Micro Edition). The programming effectiveness of MAPS and AFME with respect to the developed systems is finally discussed.**

Keywords - Mobile agent platforms; wireless body sensor networks; Java Sun SPOT; finite state machines; human activity monitoring

I. INTRODUCTION

Wireless Sensor Networks (WSNs) [1] are collection of tiny, low-cost devices with sensing, computing, storing, communication and possibly actuating capabilities. Every sensor node is programmed to interact with the other ones and with its environment, constituting a unique distributed and cooperative system aiming at reaching a global behavior and result. WSNs are a powerful technology for supporting a lot of different real-world applications, and for a demonstration it is worth noting that in the last decade this new technology has emerged in a wide range of different domains including health-care, environment and infrastructures monitoring, smart home automation, emergence management, and military support, showing a great potential for numerous other applications.

When a WSN is specifically used for being applied to the human body we deal with Wireless Body Sensor Network (WBSN) [2] which involves wireless wearable physiological sensors for strictly medical or non medical purposes. For example, they can be very effective for providing continuous monitoring and analysis of physiological or physics parameters very useful, among the others, in medical assistance, in motion and gestures detection, in emotional recognition, etc.

Unfortunately, designing such networks is not an easy work because it implies knowledge from many different areas, ranging from low-level aspects of the sensor nodes hardware and radio communication to high-level concepts concerning final user applications. Overcoming these difficulties by providing a powerful yet simple software development tool is a fundamental step for better exploiting current sensor platforms. It is quite evident that middleware supporting high-level abstraction model can be adopted for addressing these programming problems and assisting users in a fast and effective development of applications. For this reason, programming abstractions definition is one of the most fermenting research areas in the context of sensor networks, demonstrated by the several high-level programming paradigms proposed during the last years.

The main focus of this paper is to show how the agent-based programming model is an effective, easy and fast approach for developing WBSN applications, with particular emphasis on two Java-based agent platforms running on Sun SPOT sensors: MAPS (Mobile Agent Platform for Sun SPOTs) [12, 13] and AFME (Agent Factory Micro Edition) [14, 15].

The rest of this paper is structured as follows. In Section II, we first briefly review several programming paradigms for WSNs, most of which can be easily applicable in the context of WBSNs. Section III is devoted to a description of the main characteristics of a WBSN. Agent-oriented application design and implementation using MAPS and AFME are described in Section IV whereas a WBSN real-time human activity monitoring system, developed through the two aforementioned agent platforms, is described in Section V. Finally, concluding remarks are drawn.

II. PROGRAMMING PARADIGMS FOR WSN

The basic functions required by high-level programming tools are (1) to provide standard system services to easily deploy current and future applications and (2) to offer mechanisms for an adaptive and efficient utilization of system resources. Such tools embrace a wide range of software systems that can be categorized in different classes, each of which characterized by specific features so that, although most WSN applications have common requirements, many different solutions [3] have been proposed in the last years, differing on the basis of the model assumed for providing the high-level programming abstractions. On the basis of the literature so far, it emerges that none of the proposed application development methodologies can be considered the predominant one. Part of them has peculiar features specifically conceived for particular application domains but lacks in other contexts. However,

among the different programming methodologies, we believe that the exploitation of the agent-oriented programming paradigm to develop WBSN applications could provide the required effectiveness, flexibility and development easiness as demonstrated by the application of agent technology in several other key application domains [24]. In the following, several programming paradigms along with their brief descriptions are provided.

A. Database model

The database model lets users view the whole sensor network as a virtual relational distributed database system allowing a simple and easy communication scheme between network and users. Through the adoption of easy-to-use languages the latter have the ability to make intuitive queries for extracting the data of interest from the sensors. The most common way for querying networks is making use of a SQL-like language, a simple declarative-style language. This model is mainly designed to collect data streams, with the problem that it provides only approximate results and also, it is not able to support real-time applications because it lacks of time-space relations between events. TinyDB [4], Cougar [5] and SINA [6] are examples of middleware adopting this approach.

B. Macroprogramming model

This model considers the global behavior for wireless sensor network, rather than single actions related to individual nodes. The need for this approach arises when developers have to deal with WSNs constituted by a quite large number of nodes, such that the complexity resulting from the task to coordinate their actions makes applications impossible to be designed in an effective way. Macroprogramming generally have some language constructs for abstracting embedded systems' details, communication protocols, nodes collaboration, and resource allocation. Moreover, it provides mechanisms through which sensors can be divided into logical groups on the basis of their locations, functionalities, or roles. Then, programming task decreases in complexity because programmers have only to specify what kind of collaborations exist between groups, whereas the underlaying execution environment is in charge of translating these high-level conceptual descriptions into actual node-level actions. ATaG [7], Kairos [8] and Regiment [9] are based on the macroprogramming model.

C. Agent-based model

The agent-based programming model is associated with the notion of multiples, desirable lightweight, agents migrating from node to node performing part of a given task, and collaborating each other to implement a global distributed application. Mobile agents are software processes able to migrate among computing nodes by retaining their execution state. An agent could read sensor values, actuate devices, and send radio packets. The users do not have to define any per-node behaviors, but only an arbitrary number of agents with their logics, specifying how they have to collaborate for accomplishing the tasks needed to form the global application on the network. Middleware according to this model provides users with high-level constructs of a formal language for defining agents' characteristics, hiding how collaboration and mobility are actually implemented. The reasons in adopting such a model is mainly due to the need for building applications that can be reconfigured and relocated. Moreover, the key of this approach is that applications are as modular as possible to facilitate their distribution trough the network using mobile code. Agilla [10], ActorNet [11], MAPS [12, 13] and AFME [14, 15] are the main agent-based solutions for WSN.

D. Virtual machine model

Virtual machines (VM) have been generally adopted for software emulating a guest system running on top of a host real one. In the WSN context, VMs are used for allowing a vastly range of applications to run on different platforms without worrying of the actual architecture characteristics. User applications are coded with a simple set of instructions that are interpreted by the VM execution environment. Unfortunately, this approach suffers from the overhead that the instructions interpretation introduces. Maté [16] is an example of VM-based middleware.

E. Event-based model

In the context of wireless sensor networks where nodes mobility and failures are very common, the event-based middleware solutions are the effective way to support reactive and instantaneous responses to network changes. In a context where continuing data collecting and monitoring take place among a large number of nodes, a traditional request/response communication paradigm is not suitable at all, as it could be happen that some nodes (e.g. a data source node or a sink node) are not available. As a consequence, a client that continuously needs information updates could make requests without receiving any response, and this is not acceptable because energy is a scarce resource and also, this could bring to network congestion. Rather, the asynchronous event-driven communication with support for a publish/subscribe mechanism, allows a strong decoupling between sender and receiver, resulting in a more suitable approach. A client subscribes particular events so that it receives a message only when one of them occurs and also, data processing execution takes place only when necessary. Mires [17] and DSWare [18] are two examples of event-based middleware.

F. Application-driven model

Middlewares belonging to this model aim to provide services to applications according to their needs and requirements, especially for QoS and reliability of the collected data. They allow programmers to directly access the communication protocol stack for adjusting network functions to support and satisfy requested requirements. MiLAN [19] is an example of middleware based on this approach.

III. Wireless Body Sensor Networks

WSNs applied to the human body are usually called Wireless Body Sensor Networks (WBSNs) [2] involving wireless wearable physiological sensors for strictly medical or non medical purposes. WBSNs are conveying notable attention as their real-world applications aim at improving the quality of human beings life by enabling continuous and real-time non-invasive assistance at low cost. Applications where WBSNs could be greatly useful include early detection or

prevention of diseases (e.g. heart attacks, Parkinson, diabetes, asthma), elderly assistance at home (e.g. fall detection, pills reminder), e-fitness, rehabilitation after surgeries (e.g. knee or elbow rehabilitation), motion and gestures detection (e.g. for interactive gaming), cognitive and emotional recognition (e.g. for driving assistance or social interactions), medical assistance in disaster events (e.g. terrorist attacks, earthquakes, wild fires), etc.

Designing and programming applications based on WBSNs are complex tasks. That is mainly due to the challenge of implementing signal processing intensive algorithms for data interpretation on wireless nodes that are very resource limited and have to meet hard requirements in terms of wearability and battery duration as well as computational and storage resources. This is challenging because WBSNs applications usually require high sensor data sampling rates which endanger real-time data processing and transmission capabilities as computational power and available bandwidth are generally scarce. This is especially critical in signal processing systems, which usually have large amounts of data to process and transmit. WBSNs generally rely on a star-based network architecture, which is organized into a coordinator node (PDA, laptop or other) and a set of sensor nodes (see Fig. 1). The coordinator (often requiring a basestation node for the necessary communication capabilities) manages the network, collects, stores and analyzes the data received from the sensor nodes, and also can act as a gateway to connect the WBAN with other networks (e.g. Internet) for remote data access. Sensor nodes measure local physical parameters and send raw or pre-processed data to the coordinator.

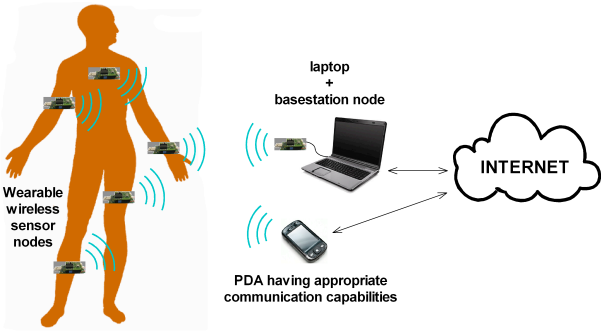

Figure 1. A typical WBSN architecture.

IV. JAVA-BASED AGENT PLATFORMS FOR WBSN

In the last years, several software frameworks have been designed for supporting WBSN. Some examples are CodeBlue [20], Titan [21], and SPINE [22] which aim at decreasing development time and improving interoperability among signal processing intensive applications based on WBSNs while fulfilling efficiency requirements. In particular, they are developed in TinyOS [23] at the sensor node side and in Java at the coordinator node side. Despite the fact that the aforementioned frameworks represent good systems for developing WBSN applications, in this paper we want to underline how mobile agents, in the context of distributed computing systems and highly dynamic distributed environments, are a suitable and effective computing paradigm for supporting the development of distributed applications, services, and protocols as demonstrated by the application of agent technology in several key application domains [24]. For more details regarding how agents can support WSN applications and address typical WSN problems, the authors remand to [27]. Among the WSN agent platforms (see Section II.C), MAPS and AFME can provide more flexibility and extendibility thanks to the Java language through which they are implemented. While MAPS was specifically conceived for Sun SPOTs [26], AFME was developed on J2ME currently supported by Sun SPOTs. In the following we describe and compare MAPS and AFME with respect to their architecture, features and programming model.

A. *MAPS: Mobile Agent Platform for Sun SPOTs*

MAPS [12, 13] is an innovative Java-based framework expressly developed on Sun SPOT technology for enabling agent-oriented programming of WSN applications. It has been defined according to the following requirement:

- Component-based lightweight agent server architecture to avoid heavy concurrency and agents cooperation models.
- Lightweight agent architecture to efficiently execute and migrate agents.
- Minimal core services involving agent migration, agent naming, agent communication, timing and sensor node resources access (sensors, actuators, flash memory, and radio).
- Plug-in-based architecture extensions through which any other service can be defined in terms of one or more dynamically installable components implemented as single or cooperating (mobile) agents.
- Use of Java language for defining the mobile agent behavior.

MAPS architecture (see Fig. 2) is based on several components interacting through events and offering a set of services to mobile agents, including message transmission, agent creation, agent cloning, agent migration, timer handling, and an easy access to the sensor node resources. In particular, the main components are the following:

- *Mobile Agent (MA)*. MAs are the basic high-level component defined by user for constituting the agent-based applications.
- *Mobile Agent Execution Engine (MAEE)*. It manages the execution of MAs by means of an event-based scheduler enabling lightweight concurrency. MAEE also interacts with the other services-provider components to fulfill service requests (message transmission, sensor reading, timer setting, etc) issued by MAs.
- *Mobile Agent Migration Manager (MAMM)*. This component supports agents migration through the Isolate (de)hibernation feature provided by the Sun SPOT environment. The MAs hibernation and serialization involve data and execution state whereas the code should already reside at the destination node (this is a current limitation of the Sun SPOTs which do not support dynamic class loading and code migration).
- *Mobile Agent Communication Channel (MACC)*. It enables inter-agent communications based on asynchronous

messages (unicast or broadcast) supported by the Radiogram protocol.

- *Mobile Agent Naming (MAN)*. MAN provides agent naming based on proxies for supporting MAMM and MACC in their operations. It also manages the (dynamic) list of the neighbor sensor nodes which is updated through a beaconing mechanism based on broadcast messages.
- *Timer Manager (TM)*. It manages the timer service for supporting timing of MA operations.
- *Resource Manager (RM)*. RM allows access to the resources of the Sun SPOT node: sensors (3-axial accelerometer, temperature, light), switches, leds, battery, and flash memory.

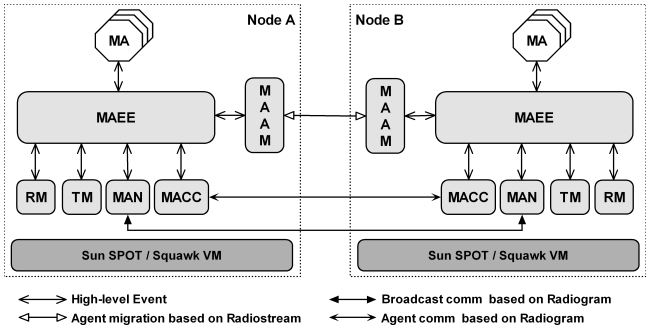

Figure 2. MAPS software architecture.

The dynamic behavior of a mobile agent (MA) is modeled through a multi-plane state machine (MPSM). Each plane may represent the behavior of the MA in a specific role so enabling role-based programming. In particular, a plane is composed of local variables, local functions, and an automaton whose transitions are labeled by Event-Condition-Action (ECA) rules *E[C]/A*, where *E* is the event name, *[C]* is a boolean expression evaluated on global and local variables, and *A* is the atomic action. Thus, agents interact through events, which are asynchronously delivered and managed by the MAEE component.

It is worth noting that the MPSM-based agent behavior programming allows exploiting the benefits deriving from three main paradigms for WSN programming: event-driven programming, state-based programming and mobile agent-based programming.

B. AFME: Agent Factory Micro Edition

AFME [14, 15] is an open-source lightweight J2ME MIDP compliant agent platform based upon the preexisting Agent Factory framework and intended for wireless pervasive systems. Thus, AFME has not been specifically designed for sensor networks but, thanks to a recent support of J2ME onto the Sun SPOT sensor platform, it can be adopted for developing agent-based WSN applications.

AFME is strongly based on the *Believe-Desire-Intention* (*BDI*) paradigm, in which agents follow a sense-deliberate-act cycle. To facilitate the creation of BDI agents the framework supports a number of system components which developers have to extend when building their applications: *perceptors*, *actuators*, *modules*, and *services*. Perceptors and actuators enable agents to sense and to act upon their environment respectively. Modules represent a shared information space between actuators and perceptors of the same agent, and are used, for example, when a perceptor may perceive the resultant effect of an actuator affecting the state of an object instance internal to the agent. Services are shared information space between agents used for data agent exchange.

The agents are periodically executed using a scheduler, and in particular four functions are performed during agent execution. First, the perceptors are fired and their sensing operations generate beliefs, which are added to the agent's belief set. A belief is a symbolic representation of information related to the agent's state or to the environment. Second, the agent's desires are identified using resolution-based reasoning, a goal-based querying mechanism commonly employed within Prolog interpreters. Third, the agent's commitments (a subset of desires) are identified using a knapsack procedure. Fourth, depending on the nature of the commitments adopted, various actuators are fired.

In AFME agents are defined through a mixed declarative/imperative programming model. The declarative Agent Factory Agent Programming Language (AFAPL), based on a logical formalism of belief and commitment, is used to encode an agent's behavior by specifying rules defining the conditions under which commitments are adopted. The imperative Java code is instead used to encode perceptors and actuators. A declarative rule is expressed in the following form:

$$b1, b2, \ldots, bn > doX;$$

where *b1*... *bn* represent beliefs, whereas *doX* is an action. The rule is evaluated during the agent execution, and if all the specified beliefs are currently included into the agent's beliefs set, the imperative code enclosed into the actuator associated to the symbolic string *doX* is executed.

The AFME platform architecture is shown in Fig. 3. It comprises a scheduler, a group of agents, and several platform services needed for supporting, among the others, agents communication and migration.

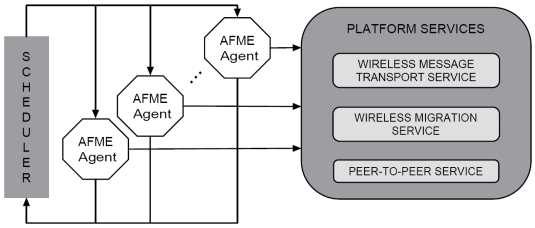

Figure 3. AFME software architecture.

To improve reuse and modularity within AFME, actuators, perceptors, and services are prevented from containing direct object references to each other. Actuators and perceptors developed for interacting with a platform service in one application can be used, without any changes to their imperative code, to interact with a different service in a different application. In the other way round, the implementation of platform services can be completely modified without having to change the actuators and the perceptors. Additionally, a same platform service may be used within two different applications to interact with a different set of actuators and perceptors. So, all system components of the AFME platform are interchangeable because they interact without directly referencing one another.

V. AN AGENT-BASED REAL-TIME HUMAN ACTIVITY MONITORING SYSTEM

The main aims of this section are (1) to demonstrate the effectiveness of agent-based platforms to support programming of WBSN applications and (2) to show how differently the two Java-based platforms, MAPS and AFME, allow defining the agent behavior in a real context. For these purposes a signal processing in-node system specialized for real-time human activity monitoring has been designed and implemented. In particular, the application is able to recognize postures (e.g. lying down, sitting and standing still) and movements (e.g. walking) of assisted livings. The architecture of the system, shown in Fig. 4, is constituted of one coordinator and two sensor nodes.

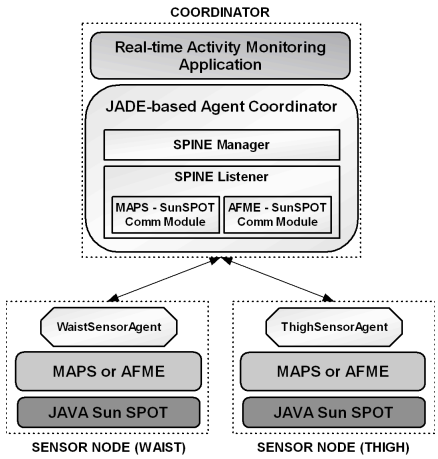

Figure 4. Architecture of the Real-time Activity Monitoring System.

The coordinator is an enhancement of the Java-based coordinator developed in the context of the SPINE project [22]. Differently from it, the new one has been re-designed around an agent developed through the JADE platform [25], which encapsulates the already implemented SPINE Manager and SPINE Listener. In particular, the SPINE Manager is used by end-user applications (e.g. the Real-time Activity Monitoring Application) for sending commands to the sensor nodes and for being notified with higher-level events and message content coming from the nodes. Then, the SPINE Manager communicates with the nodes by relying on the SPINE Listener, which integrates several sensor platform-specific SPINE communication modules. A SPINE communication module is composed of a send/receive interface and some components that implement such interface according to the specific sensor platform and that encapsulate the high-level SPINE messages into sensor platform-specific messages. In addition to the already implemented TinyOS and Z-Stack modules, not illustrated in Fig. 4, the MAPS and AFME modules have been integrated into the listener. The JADE agent coordinator also incorporates the logic for managing the synchronization of the two sensor agents.

The sensor nodes are two Sun SPOT respectively positioned on the waist and the thigh of the monitored person. In particular, MAPS or AFME runtime systems are resident on the sensor nodes and support the execution of the WaistSensorAgent and the ThighSensorAgent. The two agents have the following similar step-wise cyclic behavior:

1. Sensing on the 3-axial accelerometer sensor according to a given sampling time (ST);
2. Computation of specific features (Mean, Max and Min functions) on the acquired raw data according to the window (W) and shift (S) parameters. In particular, W is the sample size on which features are computed whereas S is the number of new acquired sample data for calculating a new feature. Usually S is set to 50% of W;
3. Features aggregation and transmission to the coordinator; Goto 1.

The agents differ in the specific computed features even though the W and S parameters are equally set. In particular, while 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, the ThighSensorAgent calculates the min value for data sensed on the X axis. The interaction diagram depicted in Fig. 5 shows the interaction among the three agents constituting the real-time system: CoordinatorAgent, WaistSensorAgent and ThighSensorAgent.

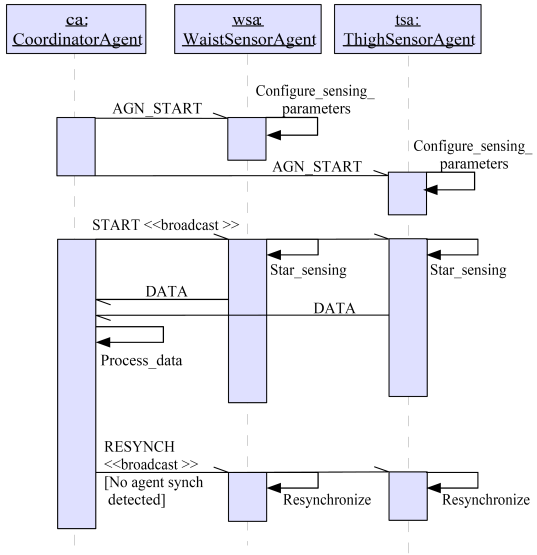

Figure 5. Agents interaction of the Real-time Activity Monitoring System.

In particular, the CoordinatorAgent first sends an AGN_START event to each sensor agent for configuring them with the sensing parameters (W, S and ST); then, it broadcasts the START event to start the sensing activity of the sensor agents. Sensor agents send the DATA event to the CoordinatorAgent as soon as features are computed. If the CoordinatorAgent detect that the agents are not synchronized anymore (i.e. agent data are received with significant time difference due to different processing tasks), it sends the RESYNCH event to resynchronize them, for not affecting the real-time monitoring (see [28] for more details).

After having described the general architecture of the system, in the following subsections we describe the implementation of the WaistSensorAgent carried out with MAPS and AFME and, then, we compare the implemented solutions.

A. *WaistSensorAgent implementation through MAPS*

As previously discussed, MAPS agents are modeled through a multi-plane state machine. The behavior of the WaistSensorAgent is specified through the 1-plane reported in Fig. 6 (the behavior of the ThighSensorAgent has the same structure but the computed features are different), whereas the most relevant parts of the Java code related to this plane are shown in Fig. 7. In particular, global variables (*GV*), local variables (*LV*), actions and related local functions (*LF*) are detailed.

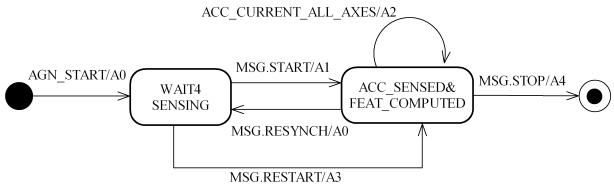

Figure 6. 1-plane behavior of the MAPS-based WaistSensorAgent model.

In the following, the agent behavior is described. The AGN_START event, issued by the coordinator agent, triggers the action A0. This action includes code for initializing some variables (see initVars function), along with (see initSensingParamsAndBuffers function) the necessary sensing parameters (W, S, ST) and buffers such as the data acquisition buffers for XYZ channels of the accelerometer sensor (windowX, windowY, windowZ) and the data buffers for features calculation (windowX4FE, windowY4FE, windowZ4FE). After that, the sensing plane goes into the WAIT4SENSING state.

The MSG.START event allows starting the sensing process by the execution of the action A1; in particular: (i) the timer is set for timing the data acquisition according to the ST parameter (see timerSetForSensing function) by using the highly precise Sun SPOT timer; (ii) a data acquisition is requested by submitting the ACC_CURRENT_ALL_AXES event and by invoking 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; in particular: (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, calculation of the features is carried out through the meanMaxMin function, and the aggregated results are sent to the base station by means of the MSG_TO_BASESTATION event appropriately constructed; (iv) the timer is reset; (v) data acquisition is finally requested.

In the ACC_SENSED&FEAT_COMPUTED state the MSG.RESYNCH might be received for resynchronization purposes; it brings the sensing plane into the WAIT4SENSING state. The MSG.RESTART brings the sensing plane back to ACC_SENSED&FEAT_COMPUTED state for reconfiguring and continuing the sensing process. The MSG.STOP eventually terminates the sensing process.

```
GV
byte timestamp;
double [] windowX4FE, windowY4FE, windowZ4FE;
String basestationAddress;

LV
int W, S, ST;
byte sampleCounter;
int nextSampleIndex;
IAT91_TC timer;
double [] windowX, windowY, windowZ;
double [] resultsX, resultsY, resultsZ;

Actions
A0: initVars();
    initSensingParamsAndBuffers(event);
A1: timerSetForSensing();
    doSensing();
A2: bufferFilling(event);
    sampleCounter++;
    nextSampleIndex=(nextSampleIndex+1)%W;
    if (sampleCounter==S){
     sampleCounter==0;
     copySensingBuffersIntoBuffersForComputingFeatures();
     computeFeatures();
     transmitFeaturesComputed();
    }
    timerReset();
    doSensing();
A3: timerDisabling();
    initVars();   A1;
A4: timerDisabling();

LF
initVars():
sampleCounter=0; nextSampleIndex=0; agent.timestamp=0;

initSensingParamsAndBuffers(Event event):
  (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();

computeFeatures():
  resultsX = meanMaxMin((WaistSensorAgent)agent.windowX4FE);
  resultsY = meanMaxMin((WaistSensorAgent)agent.windowY4FE);
  resultsZ = meanMaxMin((WaistSensorAgent)agent.windowZ4FE);

transmitFeaturesComputed():
  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);
  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);

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

Figure 7. Java code related to the WaistSensorAgent plane.

B. WaistSensorAgent implementation through AFME

The WaistSensorAgent specified trough the AFME design model is depicted in Fig. 8 whereas a code excerpt, related to the model implementation, is provide in Fig. 9. In particular, the perceive method of the SensingPerceptor, the act method of the ActivateSensorActuator and part of the SharedDataModule are detailed. The AFME agent design is constituted by the components described in the following:

- *2 Perceptors*. MTSPerceptor checks for the arrival of a new message coming from the coordinator whereas SensingPerceptor checks that the necessary features computation results are available, so that they can be sent to the coordinator.
- *3 Actuators*. ActivateSensorActuator allows activating the sensing operation, ResetActuator resets the data buffer after the reception of the resynch message, and RequestActuator is used to send a request message to the coordinator carrying the computed features.
- *Rules*. TerImplication contains the auto-generated Java description of the agent behavior rules defined into a script file (see below).
- *1 Module*. SharedDataModule contains buffers storing the data sensed from the three accelerometer channels, the feature extraction buffers and parameters, and an activation flag variable for enabling the sensing process.
- *1 Service*. RadiogramMTS represents the message transport service for data transmission to the coordinator.

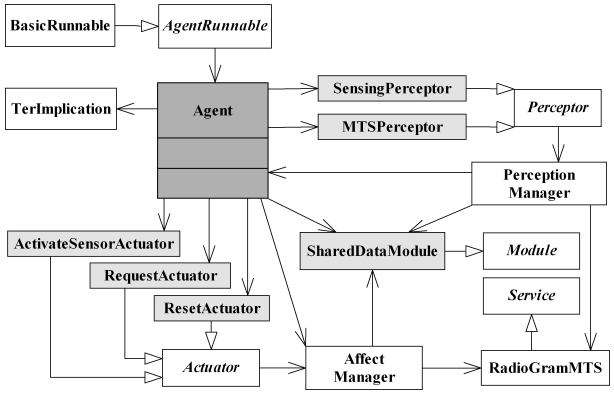

Figure 8. AFME-based WaistSensorAgent model.

The rules defining the agent behavior are the following:

```
1. message(inform, sender(BaseStation,
   addresses(BSAddress)), begin) > activateSensors(1);
2. sense(?val), !message(inform, sender(BaseStation,
   addresses(BSAddress)), resynch)  >
   request(agentID(BaseStation,
   addresses("radiogram://"+BSAddress)), ?val);
3. message(inform, sender(BaseStation,
   addresses(BSAddress)), resynch) > reset;
```

The rule (1) enables the sensor reading operation on the sensor node. In particular, this rules states that the belief generated upon the reception of an inform message, having the "begin" content and sent by the coordinator with address BSAddress, triggers the action *activateSensors*(1), which fires the ActivateSensorActuator. This actuator simply sets the activation flag on: at first, the argument of the *activateSensors* in the rule is provided to the actuator through the *action.next* method, then the call to the *aMan.actOn* (see *act* method in Fig. 9) causes the execution of the *processAction* in the SharedDatamodule (the actual flag is represented by the *activated* variable).

```
          SensingPerceptor:
....
private PerceptionManager pMan;
....
public void perceive(){
   FOS fos = pMan.perManage("shareddata", 1);
   int activated= Integer.parseInt(fos.toString());
   if(activated == 1) {
          FOS fos2= pMan.perManage("shareddata",0);
          if(fos2!=null) {
                   String fos2S= fos2.toString();
                   adoptBelief("sense("+fos2S+")");
          }
   }
}
....
          ActivateSensorActuator:
....
private AffectManager aMan;
....
public boolean act(FOS arg0) {
   FOS arg= action.next();
   aMan.actOn("shareddata", 1, null);  return true;
}
....
          SharedDataModule:
....
   // variables and data structures declaration
...
// Perceptor processing
public FOS processPer(int id)throws MalformedLogicExcep{
....
if(id==0) {
   IAccelerometer3D acceleratorSensor=
            EDemoBoard.getInstance().getAccelerometer();
   try {
          windowX[this.windowCounter]=
                            acceleratorSensor.getAccelX();
          windowY[this.windowCounter]=
                            acceleratorSensor.getAccelY();
          windowZ[this.windowCounter]=
                            acceleratorSensor.getAccelZ();
          this.windowCounter=
              ((this.windowCounter +1)%this.WINDOW_SIZE);
          this.shiftCounter++;
   }catch(IOException e){e.printStackTrace();}
   if (shiftCounter == SHIFT_SIZE){
          double meanX = meanMaxMin(this.windowX)[0];
          double meanY = meanMaxMin(this.windowY)[0];
          double meanZ = meanMaxMin(this.windowZ)[0];
          double maxY = meanMaxMin(this.windowY)[1];
          double minY = meanMaxMin(this.windowY)[2];
          String returnValues = this.timestamp+"|"+meanX+"|"+
                            meanY+"|"+meanZ+"|"+maxY+"|"+minY;
          this.timestamp=
                   ((this.timestamp +1)%this.MAX_TIMESTAMP);
          this.shiftCounter= 0;
          return FOS.createFOS(returnValues);
   }else return null;
 }else if(id==1){
   String s= String.valueOf(activated);
   return FOS.createFOS(s);
 }
....
}
// Actuator processing
public FOS processAction(int id, FOS data)
                                    throws MalformedLogicExcep{
....
   if(id==1){
          String s= data.toString();
          activated= Integer.parseInt(s);
   }
....
}
....
```

Figure 9. Java code from the AFME-based WaistSensorAgent.

The rule (2) regulates the agent behaviour during the sensing phase. If a new sense-belief is generated (i.e. features computation has been carried out on sensed data) and a message-belief, generated upon reception of the resynch message sent by the coordinator, does not exist, the request action is carried out so that a new message containing the computed features is sent to the coordinator. In particular, the sensing operation and the features computation are driven by the SensingPerceptor (see *perceive* method in Fig. 9) which, if the flag is on (*activated==1*), calls the *pMan.perManage* method with parameter "0" that in turns causes the execution of the *processPer* in the SharedDataModule (more precicely the code related to the condition *id==0*). This code performs the data reading from the accelerometer and the features computation, returning the results in the form of a belief (see *sense(?val)* in rule 2). Finally, the rule (3) allows resynchronizing the sensor agent. In particular, it states that if the belief message is generated upon reception of the resynch

message, the reset action, which re-initializes all the data structures in SharedDataModule, is executed.

C. MAPS and AFME: programming model differences

The development of MAPS and AFME agents is based on different approaches. MAPS uses state machines to model the agent behavior and directly the Java language to program conditions and actions. AFME uses a more complex model centered on perceptors, actuators, rules, modules, and services for defining the mobile agents. They are both effective in modeling agent behavior even though MAPS is more straightforward as it relies on a programming style based on state machines widely known by programmers of embedded systems. Moreover, differently from AFME, MAPS is specifically designed for WSNs and fully exploits the release 5.0 red of the Sun SPOT library to provide advanced functionality of communication, migration, sensing/actuation, timing, and flash memory storage. Finally, MAPS allows developers to program agent-based applications in Java according to its rules so no translator and/or interpreter need to be developed and no new language has to be learnt. See [28] for experimental results comparing the two agent platforms.

VI. CONCLUSION

In this paper we have presented the agent-oriented approach for high-level programming of WBAN applications. The agent approach is not only effective during the design of a WBAN application but also during the implementation phase. In particular, we have described two Java-based agent platforms, MAPS and AFME, allowing for a more rapid prototyping of sensor node code than low-level APIs which can be effectively used only by sensor node skilled programmers having knowledge of sensor drivers, communication and energy mechanisms. The higher level software abstractions provided by MAPS and AFME are suitable for a fast and easy real WBSN applications development as demonstrated by the proposed case study concerning a real-time human activity monitoring system.

ACKNOWLEDGMENT

Authors wish to thank the members of the Plasma Group at University of Calabria and in particular, Francesco Aiello, Alessio Carbone, Raffaele Gravina, and Antonio Guerrieri, for their excellent support and valuable contributions in terms of ideas, discussions and implementation efforts.

REFERENCES

[1] J. Yick, B. Mukherjee, and D. Ghosal, “Wireless sensor network survey,” Comput. Netw., vol. 52, pp. 2292-2330, August 2008.

[2] G.-Z. Yang, Body Sensor Networks, Springer 2006.

[3] Salem Hadim and Nader Mohamed, “Middleware: Middleware challenges and approaches for wireless sensor networks,” IEEE Distributed Systems Online, vol. 7, March 2006.

[4] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “Tinydb: an acquisitional query processing system for sensor networks,” ACM Trans. Database Syst., vol. 30, pp.122-173, 2005.

[5] P. Bonnet, J. Gehrke, and P. Seshadri, “Towards sensor database systems,” in MDM '01: in Proc. of the 2nd Int. Conference on Mobile Data Management, pp. 3-14, London, UK, 2001. Springer-Verlag.

[6] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen., “Sensor information networking architecture,” in Proc. of the International Workshop on Parallel Processing (ICPP), pp. 23, 2000.

[7] A. Bakshi, V. K. Prasanna, J.Reich, and D. Larner, “The abstract task graph: a methodology for architecture-independent programming of networked sensor systems,” in EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services, pp. 19-24, Berkeley, CA, USA, 2005. USENIX Association.

[8] R. Gummadi, O. Gnawali, and R. Govindan, “Macro-programming wireless sensor networks using kairos,” in International Conference on Distributed Computing in Sensor Systems (DCOSS), 2005.

[9] R. Newton, G. Morrisett, and M. Welsh, “The regiment macroprogramming system,” in Proc. of the 6th international Conference on Information Processing in Sensor Networks (IPSN '07, Cambridge, Massachusetts, USA, April 2007), pp. 489-498.

[10] C.-L. Fok, G.-C. Roman, and C. Lu, “Agilla: A mobile agent middleware for self-adaptive wireless sensor networks,” ACM Trans. Auton. Adapt. Syst., vol. 4, pp 1-26, 2009.

[11] Y Kwon, S. Sundresh, K. Mechitov, and G. Agha, "ActorNet: An Actor Platform for Wireless Sensor Networks," in Proc. of the 5th Int’l Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1297-1300, 2006.

[12] F. Aiello, G. Fortino, R. Gravina, A. Guerrieri, “A Java-based Agent Platform for Programming Wireless Sensor Networks,” The Computer Journal, pp. 1-28, 2010. doi: 10.1093/comjnl/bxq019.

[13] Mobile Agent Platform for Sun SPOT (MAPS), documentation and software at: http://maps.deis.unical.it/ (2010).

[14] C. Muldoon, G. M. P. O Hare, R.W. Collier, and M. J. O’Grady, “Agent Factory Micro Edition: A Framework for Ambient Applications,” in Proc. of Intelligent Agents in Computing Systems, Lecture Notes in Computer Science, vol. 3993, pp. 727-734, Reading, UK, Springer 2006.

[15] Agent Factory Micro Edition (AFME), documentation and software at http://sourceforge.net/projects/agentfactory/files/ (2010).

[16] P. Levis and D. Culler, “Maté: a tiny virtual machine for sensor networks,” SIGARCH Comput. Archit. News, vol. 30, pp. 85-95, 2002.

[17] E. Souto, G. Guimaraes, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz, “A message-oriented middleware for sensor networks,” in Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing (MPAC '04), pp. 127-134, New York, NY, USA, 2004.

[18] S. Li, S. Son, and J. Stankovic, “Event detection services using data service middleware in distributed sensor networks,” Telecommunication Systems, vol. 26, pp. 351-368, 2004.

[19] W.B. Heinzelman, A.L. Murphy, H.S. Carvalho, and M.A. Perillo, “Middleware to support sensor network applications,” IEEE Network Magazine Special Issue, vol. 18, pp. 6-14, January 2004.

[20] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton, “CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care,” in Proc. of MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 2004), June 2004.

[21] C. Lombriser, D. Roggen, M. Stager, and G. Troster, “Titan: A Tiny Task Network for Dynamically Reconfigurable Heterogeneous Sensor Networks,” in Verteilten Systemen (KiVS 2007), Bern, Switzerland..

[22] Signal Processing In-Node Environment (SPINE), documentation and software at http://spine.tilab.com (2010).

[23] TinyOS website, www.tinyos.net (2010).

[24] M. Luck, P. McBurney, and C. Preist, “A manifesto for agent technology: towards next generation computing,” Autonomous Agents and Multi-Agent Systems, vol. 9, pp. 203–252, November 2004.

[25] JADE, documentation and software at http://jade.tilab.com/ (2010).

[26] Sun™ Small Programmable Object Technology (Sun SPOT), http://www.sunspotworld.com/ (2010).

[27] F. Aiello, G. Fortino, A. Guerrieri, “Using mobile agents as an effective technology for wireless sensor networks,” in Proc. of the Second IEEE/IARIA International Conference on Sensor Technologies and Applications (SENSORCOMM 2008), Aug 25-31, Cap Esterel, France, 2008.

[28] F. Aiello, G. Fortino, S. Galzarano, R. Gravina, A. Guerrieri, “Signal processing in-node frameworks for Wireless Body Sensor Networks: from low-level to high-level approaches", *Wireless Body Area Networks: Technology, Implementation and Applications*. Pan Stanford publishing, 2010. To appear.