ASPINE: An Agent-oriented Design of SPINE

Fabio Luigi Bellifemine
TILAB - Telecom Italia
Via G. Reiss Romoli, 274
(10148) Torino, Italy
+39.011.228 6175
fabioluigi.bellifemine@telecomitalia.it

Giancarlo Fortino
DEIS – University of Calabria
Via P. Bucci, cubo 41C
87036 Rende(CS), Italy
+39.0984.494063
g.fortino@unical.it

ABSTRACT

Wireless sensor networks (WSNs) are emerging as powerful platforms for distributed embedding computing. Currently, flexible frameworks and middlewares are emerging to facilitate WSN application development which is usually very application-specific and involves the programming of low-level mechanisms for enabling sensing, communication and energy saving. SPINE is a domain-specific framework for distributed processing of sensed data which is currently being applied to the rapid prototyping of body sensor networks applications for human body activity recognition. In this paper we present ASPINE, an agent-oriented design of the SPINE framework. According to the agent-paradigm, each base station-side and sensor node-side component of SPINE (coordinator, sensor manager, function manager, communication manager) is designed as a software interacting agent with specific capabilities. The set of agents of a node constitute a local multi-agent system (MAS) which carries out a local goal (e.g. sensing and feature extraction on sensed data) whereas a set of cooperating nodes represents a distributed MAS pursuing a global goal (e.g. classification of human body postures). Finally, we also describe the design of an implementation of ASPINE based on MAPS, an agent platform for Java SunSPOTs.

Categories and Subject Descriptors

I.2.11 [Distributed Artificial Intelligence]: Multiagent systems; D.2.11 [Software Engineering]: Software Architectures

General Terms

Design, Language

Keywords

Wireless sensor networks, SPINE, agent programming paradigm.

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 [10]. 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.

When applied to the human body, WSNs are usually called Wireless Body Sensor Networks (WBSNs) [14]. This area is particularly dense of interest because real-world applications of WBSNs aim to improve the quality of life by enabling at low cost continuous and real-time non-invasive medical assistance. Health-care applications where WBSNs could be greatly useful include early detection or prevention of diseases, elderly assistance at home, e-fitness, rehabilitation after surgeries, motion and gestures detection, cognitive and emotional recognition, medical assistance in disaster events, etc.

However, programming WBSN applications is a complex task not only because of the intrinsic complexity of the problems but also due to the hard constraints of the wearable devices and to the lack of proper and effective software abstractions. To deal with this issue, domain-specific frameworks could be developed and effectively adopted. In particular, The SPINE Open Source project [11] aims to build a framework useful to decrease development time and improve interoperability among signal processing intensive applications based on WBSNs. The SPINE framework provides libraries of protocols, utilities and processing functions, and a lightweight Java API that can be used by local and remote applications to manage the sensor nodes or issue service requests. By providing these abstractions and libraries, that are common to most signal processing algorithms used in WBSNs for sensor data analysis and classification, SPINE also provides flexibility in the allocation of tasks among the WBSN nodes and allows the exploitation of implementation tradeoffs. Currently SPINE is implemented for several sensor platforms based on TinyOS [13] and Z-Stack [15] by using the programming paradigms offered by such platforms (event and component-based programming in TinyOS and C programming in Z-Stack) and is being effectively applied to the development of applications in the health care domain [7].

However we believe that the exploitation of the agent-oriented programming paradigm to develop WBSN applications could provide more effectiveness as demonstrated by the application of agent technology in several key application domains [3, 9]. As a consequence, in this paper, we propose an agent-oriented design of SPINE, named ASPINE, which extends the functionalities provided by SPINE and allows for a more rapid development of signal processing intensive WBSN applications in terms of agent-based systems.

The rest of this paper is organized as follow. Section 2 provides an overview of the SPINE software architectures and, in particular, describes its base station-side and sensor node-side components. Section 3 introduces the design of ASPINE, which is exemplified through a simple yet effective in-node signal processing task. Section 4 presents an initial design of ASPINE on MAPS, an agent platform for Java Sun Spots. Finally conclusions are drawn and on-going research anticipated.

2. AN OVERVIEW OF SPINE (SIGNAL PROCESSING IN-NODE ENVIRONMENT)

SPINE [6] is a framework for distributed signal processing in Wireless Body Sensor Networks (WBSN) based on the following principles:

- *Open Source*. SPINE is developed as an Open Source project to establish a broad community of users and developers that contribute to extend the framework with new capabilities and applications (http://spine.tilab.com).
- *Interoperability through high-level APIs*. SPINE provides local and remote applications with lightweight Java APIs that can be used by local and remote applications to manage the sensor nodes or send service requests, and are easily portable to devices of various capabilities, such as PCs or mobile phones, that can be used as WBSN coordinator.
- *High-level abstractions*. SPINE provides libraries of protocols, utilities and processing functions; hence, it simplifies the task of application developers by raising the level of abstraction. The layer defined by the SPINE libraries allows designers to focus on application-specific issues and program at a higher level of abstraction than TinyOS.
- *Distributed implementations of classification algorithms*. SPINE simplifies the development of applications that require complex signal processing algorithms and classifiers. For example SPINE supports distributed implementation of classification algorithms (at the base station and at the sensor nodes) to reduce the amount of data to be transmitted and save energy.

Currently the SPINE architecture centers on a star topology including one or more sensor nodes and a base station (or WBSN coordinator node). The coordinator typically manages the WBSN, collects and analyzes the data received from the sensor nodes, and acts as a gateway to connect the WBSN with wide area networks for remote data access.

In particular, SPINE has two main software components: one at the sensor node and the other one at the WBSN coordinator. In the following a description of the version 1.2 of SPINE developed for TinyOS is briefly presented.

The sensor node component, designed for TinyOS environment and written in nesC language [5], includes several utilities for signal processing such as data storage buffers, mathematical function libraries and common feature extractors used in signal processing. The SPINE framework has been structured to be platform independent and may work with different platforms running TinyOS 2.x. In this way commercial boards (such as TelosB and MicaZ) as well as proprietary ones may use SPINE functionalities, once interfaces with the sensors are implemented.

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

The main functional components of the software architecture of the current version 1.2 of SPINE are reported in Figure 1.

On the coordinator, User Applications manage a SPINE network through a lightweight and well-defined API. The surface level of SPINE lets registered applications be notified of high-level events generated by the WBSN under-control, such as discovery of new nodes, sensor data transmission, node alarms and system messages (e.g. low battery warnings). Commands issued by the user application and network generated events are respectively coded in lower-level SPINE messages and decoded in higher-level information by the SPINE Host Communication Manager which takes care of packets generation and retrieval and interfaces with the specific software components of the host platform to access the physical radio module to transmit/receive packets to/from the WBSN.

On the node side, the SPINE framework is responsible of providing developers abstractions of hardware resources such as sensors and the radio, a default set of ready-to-use common signal processing functions and, most important, a flexible and modular architecture to customize and extend the framework itself to support new physical platforms and sensors and introduce new signal processing services. In particular, the SPINE *Node Communication Manager* acts as the counterpart of the host communication manager; in addition, it possibly takes into account management policies to optimize e.g. energy consumption by an intelligent use of the radio module. The SPINE *Sensors Controller* manages and abstracts the sensors on the node platform, providing a standard interface to the diverse sensor drivers. It is responsible of sampling the sensors and storing the sensor readings in proper data buffers. The SPINE *Node Manager* is the central component, responsible of recognizing the remote requests and dispatching them to the proper components. Finally, the SPINE *Processing Manager* consists of a dispatcher for the actual processing services and a standard interface for user-defined services integration.

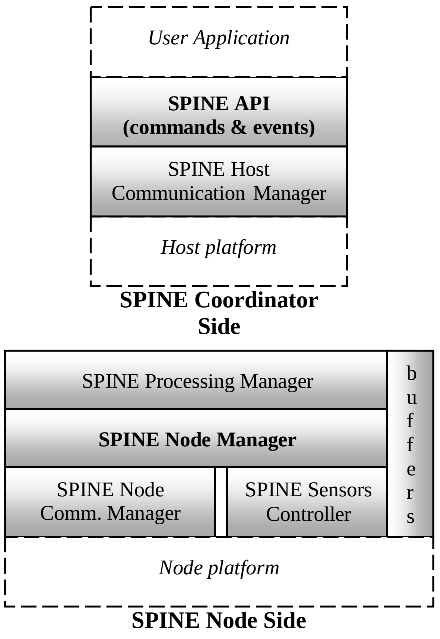


Figure 1. The software architecture of SPINE 1.2.

3. AN AGENT-ORIENTED DESIGN OF SPINE: ASPINE

The SPINE architecture overviewed in section 2 was designed by using an agent-based approach. The so obtained agent-oriented design of SPINE (or ASPINE) adheres to the organization of the SPINE architecture so consisting of base station-side and sensor-node-side agents.

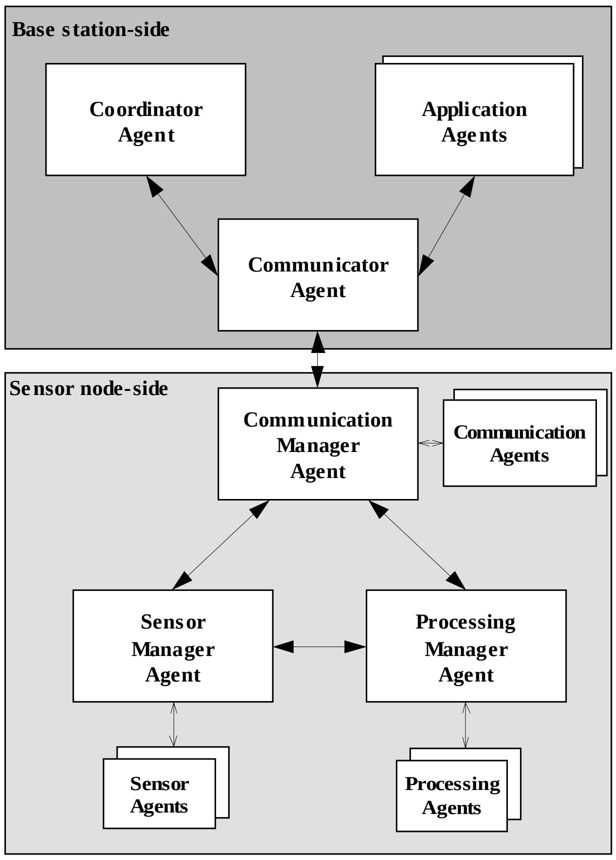


Figure 2. The ASPINE architecture.

Figure 2 shows the architecture of ASPINE through a class diagram. In particular, the following core agents are defined:

Base station-side

- The *CoordinatorAgent* is responsible for managing the set of nodes of the sensor network under control. Management involves configuring and monitoring nodes;
- The *ApplicationAgents* are agents implementing application-specific or domain-specific logics;
- The *CommunicatorAgent* allows the *CoordinatorAgent* and the *ApplicationAgents* to interact with the sensor nodes through an efficient over-the-air application-level protocol.

Sensor node-side

- The *SensorManagerAgent* manages the sensor/actuator resources of the node through specific *SensorAgents* able to interact with specific sensors (temperature, light, accelerometer, etc).
- The *CommunicationManagerAgent* manages the communication with the CommunicatorAgent and among the *CommunicationManagerAgent*s, located at different sensor nodes, by means of specific *CommunicationAgents*.
- The *ProcessingManagerAgent* supports one or more local processing tasks or parts of global processing tasks through *ProcessingAgents*. They are able to perform computation on sensed data (e.g. feature extraction) and data aggregation.

3.1 In node signal processing task based on ASPINE: an example

An example of in-node signal processing task is portrayed in Figure 3 by means of a data-flow model based on subtasks. In particular, the sensed data periodically produced by the sensing subtask acting on a 3-axial accelerometer are split for the computation of the features Mean on all three axes (XYZ), and Min and Max on axis X. Each triple of computed features (<Mean(AccXYZ), Min(AccX), Max(AccX)>) are aggregated by the aggregation subtask (Aggr) and sent to the coordinator node by the data transmission subtask (Sender) as soon as aggregated data are available.

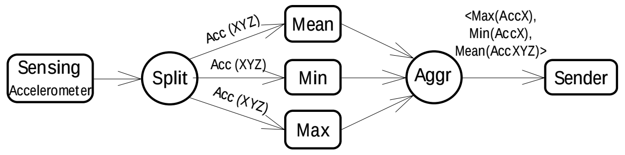


Figure 3. Data-flow-based model of an in-node signal processing task.

The ASPINE design of the model of Figure 3 is shown in Figure 4.

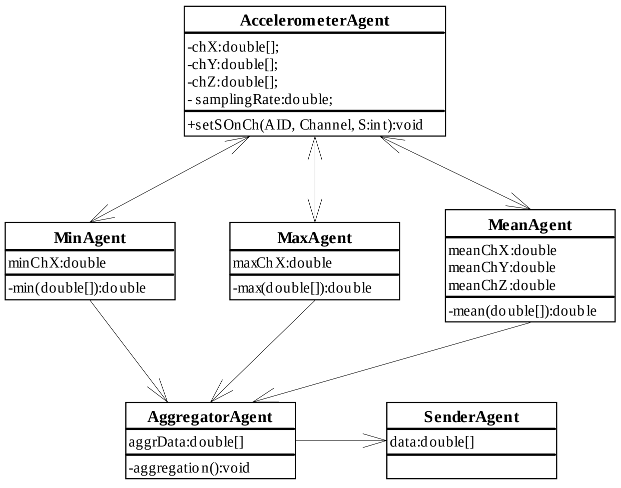


Figure 4. ASPINE-base design of the example in-node signal processing task.

The AccelerometerAgent interacts with the accelerometer sensor and, according to the set sampling rate (samplingRate), acquires one sample per channel (X, Y, Z) and stores them into the corresponding buffers chX, chY, chZ. Once S samples are acquired the AccelerometerAgent passes them to the MinAgent, MaxAgent and MeanAgent. These relationships are created through the method setSOnCh(AID, Channel, S), where AID is the agent identifier, Channel refers to the channels to be considered, and S is the number of samples. In this case, all agents are based on the same S but on different channels. In particular, MinAgent, MaxAgent and MeanAgent receive the last acquired S samples respectively from the chX buffer, from the chX buffer, and from the chX, chY, and chZ buffers. Upon reception of such data, the agents compute their respective functions and pass the results to the AggregatorAgent. This waits for the aggregation of the data triple aggrData=<minChX, maxChX, <meanChX, meanChY, meanChZ>> and passes it to the SenderAgent, a specific CommunicationAgent, which, in turns, transmits it to the CommunicatorAgent located at the base station.

4. TOWARDS AN ASPINE IMPLEMENTATION BASED ON MAPS

In this section we describe a design of ASPINE through the MAPS framework [1] which provides a real basis for its full-fledged Java implementation.

4.1 An overview of MAPS

MAPS (Mobile Agent Platform for Sun SPOTs) is an innovative Java-based framework for wireless sensor networks based on Sun SPOT technology [12] which enables agent-oriented programming of WSN applications.

MAPS has been appositely defined for resource-constrained sensor nodes; in particular the requirements on which it is based are the following:

(i) *Component-based lightweight agent server architecture*. This implies the avoidance of heavy concurrency models and, therefore, the exploitation of cooperative concurrency or single-threading to run agents.

(ii) *Lightweight agent architecture* to efficiently execute and migrate agents.

(iii) *Minimal core services*. The main core services are: 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 code, data and execution state. The sensing capability access service allows agents to access to the sensing capabilities of the sensor node, and, more generally, to 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 which allows local and remote one-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 agent or cooperating mobile agents.

(v) *Java* as programming language for agents.

The architecture of MAPS is reported in Figure 5. In particular:

- The mobile agent execution engine (MAEE) is the component which 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 the mobile agent (MA) 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.
- The mobile agent migration manager (MAMM) 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; (ii) receive a message containing a serialized MA, deserialize and activate it. The agent serialization format includes code, data and execution state.
- The mobile agent communication channel (MACC) enables inter-agent communications based on asynchronous messages. Messages can be unicast, multicast or broadcast.
- The mobile agent naming (MAN) provides agent naming based on proxies to support the MAMM and MACC components in their operations. The MAN also manages the (dynamic) list of the neighbor sensor nodes.
- The timer manager provides the timer service which allows for the management of timers to be used for timing MA operations.
- The resource manager (RM) provides access to the sensor node resources: sensors/actuators, battery, and flash memory.

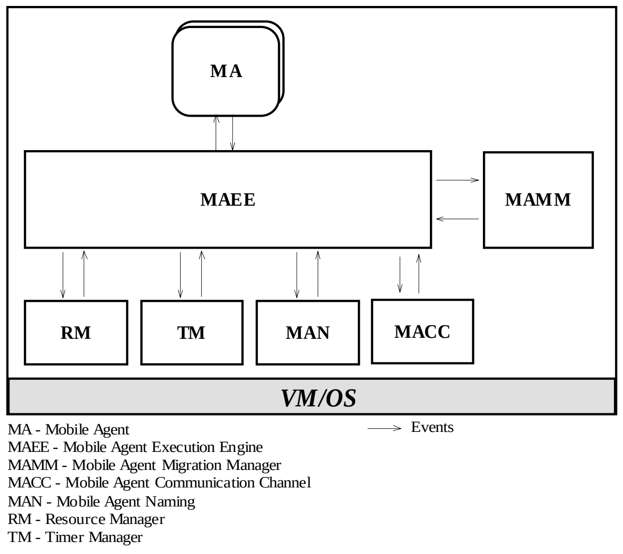

MA
MAEE
MAMM
RM
TM
MAN
MACC
VM/OS
MA - Mobile Agent
Events
MAEE - Mobile Agent Execution Engine
MAMM - Mobile Agent Migration Manager
MACC - Mobile Agent Communication Channel
MAN - Mobile Agent Naming
RM - Resource Manager
TM - Timer Manager

Figure 5. The MAPS architecture.

4.1.1 Programming abstractions of MAPS

The main programming abstractions of MAPS are *Agents* and *Events*. While events formalize interaction among components and agents, agents are the active entities whose behavior is modeled as a multi-plane state machine (MPSM). Thus MPSM-based agent behavior programming allows exploiting the benefits deriving from three paradigms for WSN programming: event-driven programming [5], state-based programming [8] and mobile agent-based programming [4, 2].

The communication among agents, between agents and system components, and, sometimes, among components are based on Event objects. An Event object is composed of:

- sourceID, which is the agent/component identifier of the event source;
- targetID, which is the agent/component identifier of the event target;
- typeName, which represents the name of the event types which are grouped according to their specific function;
- params, which include the event data organized as a chain of pairs <key, value>;
- durationType, which specifies the event duration. It can assume the following three values:
  - NOW, for instantaneous events;
  - FIRST_OCCURRENCE, for events which wait for a specific value to occur;
  - PERMANENT. In this case, the event is sent every time that values set in the event parameters are reached.

The agent behavior consists of:

- *Global variables (GV)* which represent the data of the MA including the MA identity.
- *Global functions (GF)* which consist of a set of supporting functions which can access GV but cannot invoke neither core primitives nor other functions.
- *Multi-plane State Machine (MPSM)* which 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:
  - *Local variables (LV)* which represent the local data of a plane.
  - *Local functions (LF)* which consist of a set of local plane supporting functions which can access LV but cannot invoke neither core primitives nor other functions.
  - *ECA-based Automata (ECAA)* which 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 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 (Figure 6; see [1] for more details) 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.
- *Event dispatcher (ED)* which dispatches the event delivered by the MAEE to one or more planes according to the events 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.

```
send(SourceMA, TargetMA, EventName, Params, Local)
SourceMA  = id of the invoking 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)
BackEvent  = event notifying the input captured from the
switch

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
```

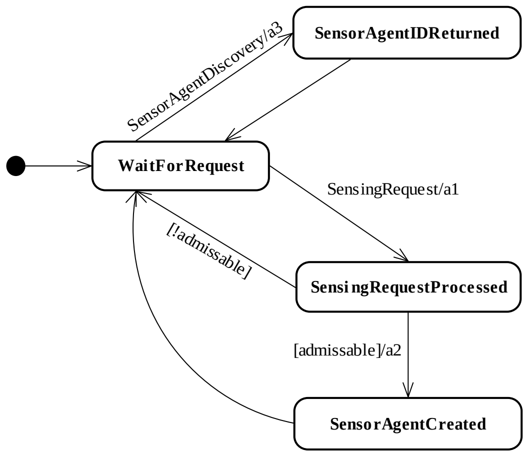
Figure 6. The prototypal core primitives.

4.2 A MAPS-based design of ASPINE

In the following we describe the prototypical behaviors of the sensor node-side agents of ASPINE (see section 3) designed through MAPS.

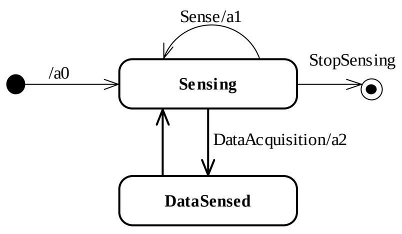
The behavior of the SensorManagerAgent consists of the state machine shown in Figure 7. It basically handles two events: SensingRequest and SensorAgentDiscovery. A SensingRequest can be admissible or not depending on the requested sensors: whether or not it is already in use (see action a1). In the former case, the SensorManagerAgent creates a SensorAgent with the sensing configuration parameters passed in the SensingRequest event (see action a2). A DiscoverySensorAgent event requests the identifier of the SensorAgent, if existing, attached to a given sensor type (see action a3).

The behavior of the SensorAgent is described by the state machine depicted in Figure 8. In particular, the Sense event is driven by a timer set according to the sensing sampling rate (see action a0). When the Sense is received, the sense operation is issued (see action a1) and, after data acquisition, sensed data are buffered into the data acquisition buffer/s of the SensorAgent (see action a2).



```
a1: SensingRequest e = (SensingRequest)evt;
    if (sensorAgents.get(e.getSensorType())==null)
       admissable=true;
    else admissable=false;
a2: create(genAgentID(),“SensorAgent”, e.getConfParams(),
           local);
a3: SensorAgentType sensorType = (DiscoverySensorAgent)
                                evt.getSensorAgentType();
AID sensorAgentID=sensorAgents.get(sensorType);
    send(self(),(DiscoverySensorAgent)evt.getSource(),
        ”AgentID”, <SAI=sensorAgentID>, true);
```

Figure 7. The SensorManagerAgent behavior.

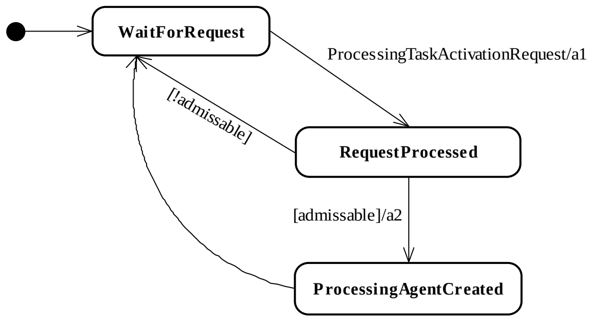


```
a0: Sense timer =new Sense(self(), self(),
                        Event.TMR_EXPIRED, Event.NOW);
     timerID = setTimer(self(), samplingTime, timer);
a1: DataAcquisition dataEvt = new DataAcquisition(self(),
                   self(), Event.SENSOR_TYPE, Event.NOW);
    sense(dataEvt);
a2: DataAcquisition e = (DataAcquisition)evt;
    buffer(e.getData());
```

Figure 8. The SensorAgent behavior.

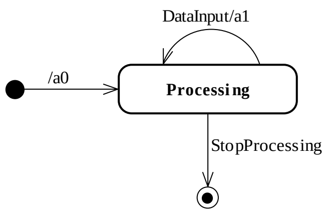
The behavior of the ProcessingManagerAgent is described by the state machine depicted in Figure 9. When the ProcessingTaskActivationRequest arrives, the ProcessingManagerAgent interprets the request and, if the request is admissible, creates a ProcessingAgent and links it to its input and output agents (i.e. agents providing data input to and receiving data output from the created ProcessingAgent – see the example of Figure 4).

The state machine of the behavior of the ProcessingAgent is reported in Figure 10. After initialization (see action a0), the ProcessingAgent is able to receive DataInput events from its input agents and process them (see action a1). After processing, the output is sent to the attached data output ProcessingAgents.



```
a1: ProcessingTaskActivationRequest e =
                      (ProcessingTaskActivationRequest)evt;
    admissable=check(e.getRequest());
a2: String agentType=interpret(e.getRequest());
    AID agID = genAgentID();
    create(agID,agentType, e.getConfParams(), local);
    link(agID);
```

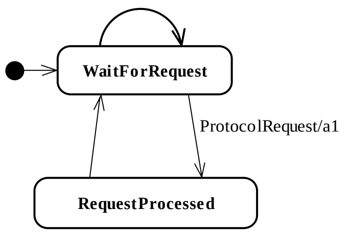
Figure 9. The ProcessingManagerAgent behavior.



```
a0: initProcessing();
a1: DataInput e = (DataInput)evt;
    process(e.getData());
```

Figure 10. The ProcessingAgent behavior.

The basic behavior of the CommunicationManagerAgent is described by the state machine depicted in Figure 11. The ProtocolRequest event encapsulates the packet of the interaction protocol with the CommunicatorAgent at the base-station; once the event is received the CommunicationManagerAgent processes it according to the protocol (see action a1) or routes it to the target manager agent, if it is not able to handle it. A ProtocolRequest involving data transmission from the node to the base station or to another node can also be requested by a SenderAgent.



```
a1: ProtocolRequest e=(ProtocolRequest)evt;
    process(e.getRequestType());
```

Figure 11. The CommunicationManagerAgent behavior.

5. CONCLUSION

In this paper we have presented ASPINE, an agent-based design of the SPINE framework. We strongly believe that the agent-oriented paradigm can simplify the redesign of SPINE to make it more flexible and effective. Moreover, with respect to the current version 1.2 of SPINE which relies on a star-based topology network, ASPINE is designed to be exploited on more general WSN topologies as agent on the sensor nodes can interact not only with agents at the base station but also among them in single-hop and multi-hop scenarios.

Current efforts are geared at implementing ASPINE through MAPS, an agent framework for Java SunSpot-based wireless sensor platforms so as to provide a full-fledged Java implementation of ASPINE at node side. Moreover, Jade and Jade Leap [3] have been considered as frameworks to be used respectively on PC and PDA-based gateways to implement ASPINE at the coordinator node.

Finally, the human activity monitoring application developed in [6], which allows recognizing postures (sitting, lying, standing) and movements (walking and falling) of individuals, is being reverse engineered to encompass sensor nodes based on ASPINE. This will offer an interesting experimentation testbed for ASPINE.

6. ACKNOWLEDGMENTS

Authors wish to thank the SPINE and MAPS research teams (particularly Roberta Di Giannantonio, Marco Sgroi, Antonio Guerrieri, Raffaele Gravina, Francesco Aiello, Alessio Carbone, Stefano Galzarano) for their precious contributions to ASPINE in terms of ideas, discussions and on-going implementation efforts. The SPINE project is partially funded by Telecom Italia.

7. REFERENCES

[1] F. Aiello, G. Fortino, R. Gravina, A. Guerrieri, "MAPS: a Mobile Agent Platform for Java Sun SPOTs," In *Proceedings of the 3rd International Workshop on Agent Technology for Sensor Networks* (ATSN-09), jointly held with the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-09), 12th May, Budapest, Hungary, 2009.

[2] 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.

[3] F.L. Bellifemine, G. Caire, D. Greenwood, "Developing Multi-Agent Systems with JADE," Wiley, 2007.

[4] C-L Fok, G-C Roman, C Lu, "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications," In Proc. of the 24th Int'l Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, June 6-10, 2005, pp. 653-662.

[5] 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. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (San Diego, California, USA, June 09 - 11, 2003). PLDI '03. ACM, New York, NY, 1-11.

[6] R. Gravina, A. Guerrieri, G. Fortino, F. Bellifemine, R. Giannantonio, M. Sgroi, "Development of body sensor network applications using SPINE," In *Proc. of. IEEE International Conference on Systems, Man, and Cybernetics* (SMC 2008), Singapore, Oct. 12-15, 2008.

[7] S. Iyengar, F. Tempia Bonda, R. Gravina, A. Guerrieri, G. Fortino, A. Sangiovanni-Vincentelli, "A Framework for Creating Healthcare Monitoring Applications Using Wireless Body Sensor Networks", In the Proc. of the 3rd International Conference on Body Area Networks (BodyNets'08), Tempe (AZ), USA, Mar. 13-15, 2008.

[8] O. Kasten, K. Römer, "Beyond event handlers: programming wireless sensors with attributed state machines," In Proc. of the 4th Int'l symposium on Information processing in sensor networks, April 24-27, 2005, Los Angeles, CA.

[9] M. Luck, P. McBurney, C. Preist, "A Manifesto for Agent Technology: Towards Next Generation Computing," Autonomous Agents and Multi-Agent Systems 9(3), pp. 203-252, 2004.

[10] K. Sohraby, D. Minoli, T. Znati, "Wireless Sensor Networks: technology, protocols, and applications", Wiley, 2007.

[11] SPINE (Signal Processing In-Node Environment), documentation and software, http://spine.tilab.com (2009)

[12] Sun Spots, documentation and software, http://www.sunspotworld.com/ (2009)

[13] TinyOS, documentation and software, http://www.tinyos.net (2009).

[14] Guang-Zhong Yang, "Body Sensor Networks", Springer, 2006.

[15] Z-Stack (ZigBee Protocol Stack), Texas Instruments, documentation and software, http://focus.ti.com/docs/toolsw/folders/print/z-stack.html (2009)