ISBN 978-83-60810-27-9
ISSN 1896-7094

Java-based Mobile Agent Platforms for Wireless Sensor Networks

Francesco Aiello, Alessio Carbone, Giancarlo Fortino*, Stefano Galzarano
DEIS, Università della Calabria, Via P.Bucci cubo 41c, 87036 Rende (CS), Italy
Email: {faiello, acarbone, galzarano}@si.deis.unical.it, g.fortino@unical.it

***Abstract*—This paper proposes an overview and comparison of mobile agent platforms for the development of wireless sensor network applications. In particular, the architecture, programming model and basic performance of two Java-based agent platforms, Mobile Agent Platform for Sun SPOT (MAPS) and Agent Factory Micro Edition (AFME), are discussed and evaluated. Finally, a simple yet effective case study concerning a mobile agent-based monitoring system for remote sensing and aggregation is proposed. The proposed case study is developed both in MAPS and AFME so allowing to analyze the differences of their programming models.**

***Keywords*: Mobile agent platforms, wireless sensor networks, Java Sun SPOT, finite state machines, intentional agents.**

I. Introduction

Due to recent advances in electronics and communication technologies, Wireless Sensor Networks (WSNs) are currently emerging as one of the most disruptive technologies enabling and supporting next generation ubiquitous and pervasive computing scenarios. A WSN is a network of RF transceivers, sensors, machine controllers, microcontrollers, and user interface devices with at least two nodes communicating by means of wireless transmissions. 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 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 run-time as well as ease software programming at application, middleware and network layer. Among the programming paradigms proposed for the development of WSN applications [1], the mobile agent-based paradigm [2], 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 application, middleware and network levels.

*Corresponding author

In this paper we present the currently available mobile agent platforms for WSNs which are based either on TinyOS [3] or on Java Sun SPOT [4]. In particular, we focus on MAPS and AFME, the only two available Java-based platforms, by analyzing their architecture, programming model and core performances. Finally they are also exemplified through an effective case study showing the different programming mechanisms of the two platforms.

The rest of this paper is organized as follows. Section II introduces mobile agents, presenting their characteristics and benefits by focusing on the WSN context; moreover, work related to currently available mobile agent systems for WSNs is discussed. Section III describes two Java-based agent platforms (MAPS and AFME) which are also compared with respect to their architecture, programming model and performance. Section IV presents an example application for showing how differently agent behavior can be defined through MAPS and AFME. Finally, conclusions are drawn and on-going research efforts delineated.

II. Mobile Agents for Wireless Sensor Networks

In the context of distributed computing systems and highly dynamic distributed environments, mobile agents are a suitable and effective computing paradigm for supporting the development of distributed applications, services, and protocols [1]. Mobile agents are software processes able to migrate among computing nodes by retaining their execution state (strong mobility). In their seminal paper [2], Lange and Oshima advertised at least seven good reasons for using mobile agents in generic distributed systems. In the following we discuss them by focusing on the WSN context.

1. Network load reduction. Mobile agents are able to access remote resources, as well as communicate with any remote entity, by directly moving to their physical locations and interacting to them locally so that to save bandwidth resources. A mobile agent incorporating data processing capabilities can migrate to a sensor node, perform the needed operations on the sensed data and transmit the results to a sink

node. This is more desirable, rather than a periodic transmission of raw sensed data from the sensor node to the sink node and the computation of data processing on the latter.
2. *Network latency overcoming*. An agent provided with proper control logic may move to a sensor/actuator node to locally perform the required control tasks. This overcomes the network latency which will not affect the real-time control operations also in case of lack of network connectivity with the base station.
3. *Protocol encapsulation*. Suppose that a specific routing protocol supporting multi-hop paths should be deployed in a given zone of a WSN. A set of cooperating mobile agents encapsulating the routing protocol can be dynamically created and distributed into the proper sensor nodes without any regard for standardization matter. Also in case of protocol upgrading, a new set of mobile agents can easily replace the old one at run-time.
4. *Asynchronous and autonomous execution*. These are distinctive properties of mobile agents and very important in dynamic environments like WSNs where connections may not be stable and network topology may change rapidly. A mobile agent, upon a request, can autonomously travel across the network to gather needed information "node by node" or to carry out the programmed tasks and, finally, can asynchronously report the results to the requester.
5. *Dynamic adaptation*. Mobile agents can perceive their execution environment and react autonomously to changes. This behavioral dynamic adaptation is well suited for operating on long-running systems like WSNs where environment conditions are very likely to change over time.
6. *Orientation to heterogeneity*. Mobile agents can act as wrappers among systems based on different hardware and software. This ability can well fit the need for integrating heterogeneous WSNs supporting different sensor platforms or connecting WSN and other networks (like IP-based networks). An agent may be able to translate requests coming from a system into specific suitable requests to submit to another different system.
7. *Robustness and fault-tolerance*. The ability of mobile agents to dynamically react to unfavorable situations and events (e.g. low battery level) can lead to a better robust and fault tolerant distributed systems (e.g. by migrating all executing agents to an equivalent sensor node so that to continue their operations).

Mobile agents are supported by mobile agent systems (MASs) which basically provide an API for developing agent-based applications, and an agent server able to execute agents by providing them with basic services such as migration, communication, and node resource access. Developing flexible and efficient MASs for WSNs is a challenging and very complex task due to the currently available resource-constrained sensor nodes and related operating systems. Very few MASs for WSNs have been to date proposed and actually implemented. In the following, we discuss Agilla and actorNet, the most significant available research prototypes based on TinyOS

Agilla [5] is an agent-based middleware developed on TinyOS [3] and supporting multiple agents on each node. As shown by its software architecture (see Fig. 1), Agilla provides two fundamental resources on each node: a tuplespace and a neighbors list. The tuplespace represents a shared memory space where structured data (tuples) can be stored and retrieved, allowing agents to exchange information in a decoupling way. A tuplespace can be also accessed remotely. The neighbors list contains the address of all one-hop nodes, needed when an agent has to migrate.
Agents can migrate carrying their code and state, but do not carry their tuples locally stored on a tuplespace. Packets used for nodes communication (e.g. for agent migration/cloning, remote tuples accessing) are very small to minimize messages loss, whereas retransmission techniques are also adopted.

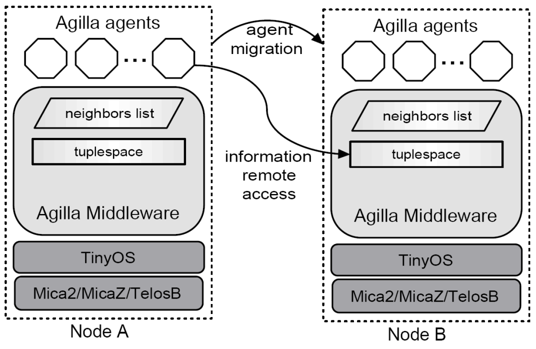

Fig. 1 Agilla software architecture

ActorNet [6] is an agent-based platform specifically designed for Mica2/TinyOS sensor nodes. To overcome the difficulties in allowing code migration and interoperability due to the strict coupling between applications and sensor node architectures, actorNet exposes services like virtual memory, context switching, and multi-tasking. Thanks to these features, it effectively supports agents programming by providing a uniform computing environment for all agents, regardless of hardware or operating system differences. The actorNet architecture is depicted in Fig. 2. The actorNet language used for high-level agent programming, has a syntax and a semantic similar to those of Scheme [7] with proper instructions extension.

Both Agilla and actorNet are designed for TinyOS which is based on the nesC language that is not an object-oriented language but an event- and component-based extension of C. The Java language, through which Sun SPOT [4] and Sentilla JCreate [18] sensors can be programmed, due to its object-oriented features, could provide more flexibility and extendibility for an effective implementation of agent-based platforms. The most significant available Java-based agent platforms for WSNs are MAPS [8, 9, 10] and AFME [11, 12, 13], which are discussed in Section III.

In Table 1, a comparison among the aforementioned agent platform with respect to 7 characteristics (migration, multitasking, communication model, programming language, remote configuration, intentional agents, and sensor platforms) is reported.

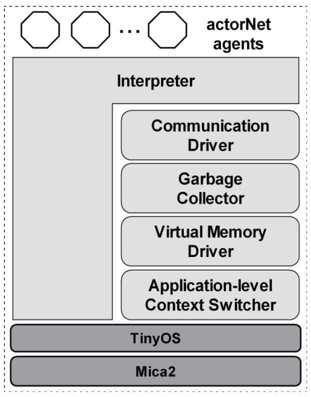

Fig. 2 ActorNet software architecture

TABLE 1
COMPARISON AMONG DIFFERENT WSN MASS

	Agilla	*actorNet*	*MAPS*	*AFME*
Migration	Y	Y	Y	Y
Multitasking	Y	Y	Y	Y
Communication Model	tuple space	messages	messages	messages
Programming Language	proprietary ISA	Scheme-like	Java	Java
Agent Model	Assembler-like	Functional	Finite State Machine	BDI
Intentional Agents	N	N	N	Y
Sensor Platforms	Mica2, MicaZ, TelosB	Mica2	Sun SPOT	Sun SPOT

Agent migration and multitasking, which allows for the execution of multiple agents on the same node, is supported by all the systems. The communication model of Agilla is centered on local tuplespace where agent can asynchronously insert tuples and take tuples left by other agents. Conversely the communication model of the other systems is based on (unicast and broadcast) message-passing. The programming language and model is different among the systems. Agilla is based on a proprietary low-level language composed of an assembler-like instruction set which makes programming of complex agents very difficult. ActorNet is based on a functional Scheme-like language whereas MAPS and AFME on the Java language. Indeed, MAPS uses a finite state machine model to define agent behaviour whereas AFME employs a more complex BDI-like model. Intentional agents are therefore only offered by AFME. Agilla and actorNet run on motes; in particular Agilla on Mica2, MicaZ, and TelosB, whereas actorNet currently only on Mica2. On the contrary MAPS and AFME are based on Sun SPOTs.

III. JAVA-BASED MOBILE AGENT PLATFORMS FOR WSNS

A great variety of Java-based agent platforms have been to date developed for the standard Java virtual machine atop conventional distributed systems. However, in the context of wireless sensor networks, only two Java-based platforms are currently available: MAPS (Mobile Agent Platform for Sun SPOT) and AFME (Agent Factory Micro Edition). While MAPS was specifically conceived for Sun SPOTs, AFME was developed for J2ME enabled PDAs and then ported onto Sun SPOTs. In the following we describe and compare MAPS and AFME with respect to their architecture, programming model and performance.

A. Mobile Agent Platform for Sun SPOTs

MAPS [8, 9, 10] 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) agent/s.
- Use of Java language for defining the mobile agent behavior.

MAPS architecture (see Fig. 3) is based on components which interact through events and offer a 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. In particular, the main components are:
- *Mobile Agent (MA)*. MAs are the basic high-level component defined by user for constituting 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 [4]. 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.

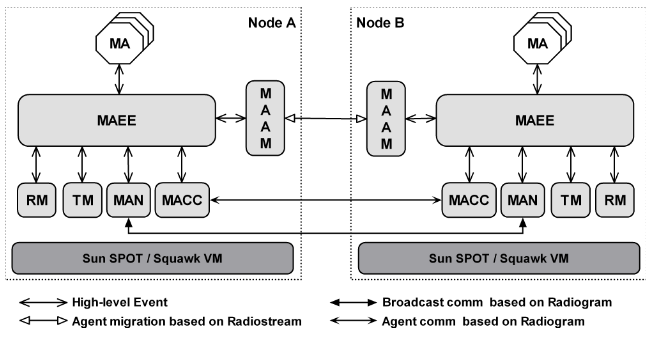

Fig. 3 MAPS software architecture

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 based on the 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. Agent Factory Micro Edition

AFME [11, 12, 13] is an open-source lightweight J2ME MIDP compliant agent platform based upon the preexisting Agent Factory framework [14] 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 [15], 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 four functions are performed when an agent is executed. 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, ..., 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. 4. It comprises a scheduler, a group of agents, and several platform services needed for supporting, among the others, agents communication and migration.

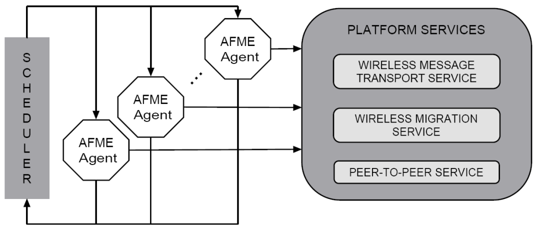

Fig. 4 AFME platform architecture

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 modify the actuators and the perceptors. Additionally, the 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.

C. MAPS vs. AFME: A comparison

Both MAPS and AFME offer similar services for developing WSN agent-based application. Nevertheless, the definition of agents is based on different approaches. MAPS uses state machines to model the agent behavior and directly the Java language to program guards and actions. AFME uses a more complex model centered on perceptors, actuators, rules, modules, and services that define the agent behavior. 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. An example is represented by the implementation of mobile agents through isolates, whose migration mechanism is directly offered by the SPOT Squawk JVM. Isolates are not used in AFME due to its employment as a more generic agent platform for CLDC-compliant devices. Apart from the agent implementation mechanism adopted, both platforms suffers from the current limitation of the Sun SPOTs which do not allow dynamic class loading, preventing from the possibility to support code migration (i.e. any classes required by the agent must already be present at the destination). 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.

To evaluate and compare the performance of MAPS and AFME two benchmarks have been defined according to [16] for the following mechanisms:

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 Back and Forward (B&F)*, in which both request and reply contain the same amount of data; (ii) *data B*, in which only the reply contains data. Comparison results are shown in Fig. 5. For agents with light data payload AFME performs better than MAPS; however, when the agent data payload overtakes 700 bytes MAPS starts performing better in the case *data B&F*.

Agent migration. The agent migration time is calculated for agent ping-pong among two single-hop-distant sensor nodes. Migration times are computed by varying the data cargo of the ping-pong agent. The obtained migration times are high due to the slowness of the SquawkVM operations supporting the migration process. Comparison results are shown in Fig. 6. AFME retains a higher performance migration mechanism.

IV. An Agent-Based Application Example

To demonstrate the effectiveness of agent-based platforms to support programming of WSN applications, a simple and exemplificative remote monitoring application has been developed. In particular, this section will show how differently the two Java-based platform MAPS and AFME allow defining the agent behavior. The proposed application example

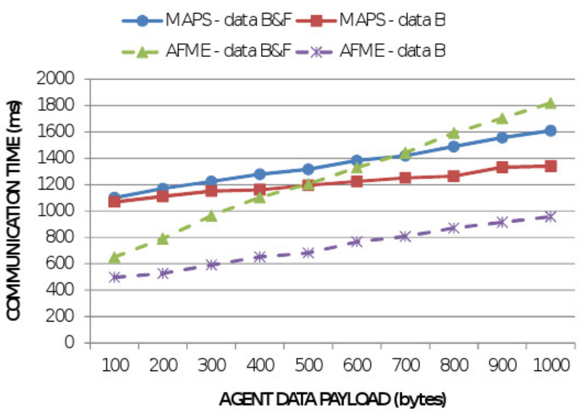

Fig. 5 Agent communication time comparison

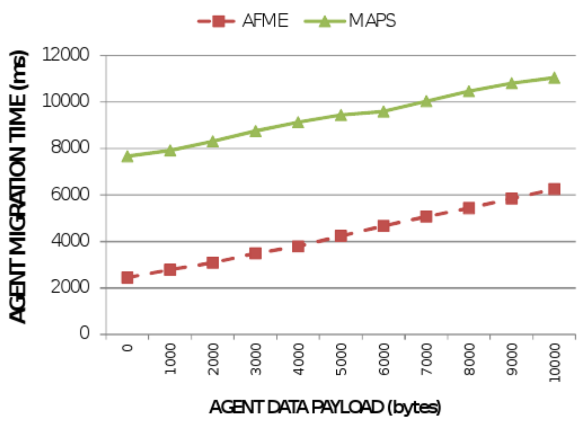

Fig. 6 Agent migration time comparison

involves two sensor nodes and consists in the following three interacting agents:

- *DataCollectorAgent*, which collects data related to the Sun SPOT node sensors (accelerometer, temperature, light);
- *DataMessengerAgent*, which carries collected sensed data from the sensing node to the basestation;
- *DataViewerAgent*, which displays the received collected data.

The sequence of interactions among the three defined agents is shown in Fig. 7 through an M-UML sequence diagram [17]. The application execution is driven by the user by pressing a switch on the Sun SPOT on which the DataViewerAgent is running. Upon the user event the DataViewerAgent sends a remote message to the DataCollectorAgent (running on the other node) for starting its sensing operations. The agent therefore starts an internal timer to a particular value to begin its collecting activity: on timer expiration the agent acquires data from the onboard node sensors and collects them. As soon as the agent has acquired *numData* samples, it calculates a set of features (e.g. max, min and mean) for each of the sensor data types. Afterwards, the DataCollectorAgent creates the DataMessengerAgent which, carrying the computed features migrates to the DataViewerAgent node for data visualization. In case the user presses the switch on the node where the DataCollectorAgent is running, the agent sends an instantaneous message having the values of the last computed features. Finally, the remote monitoring activity terminates when the user presses again a switch of the Sun SPOT on which the DataViewerAgent is running.

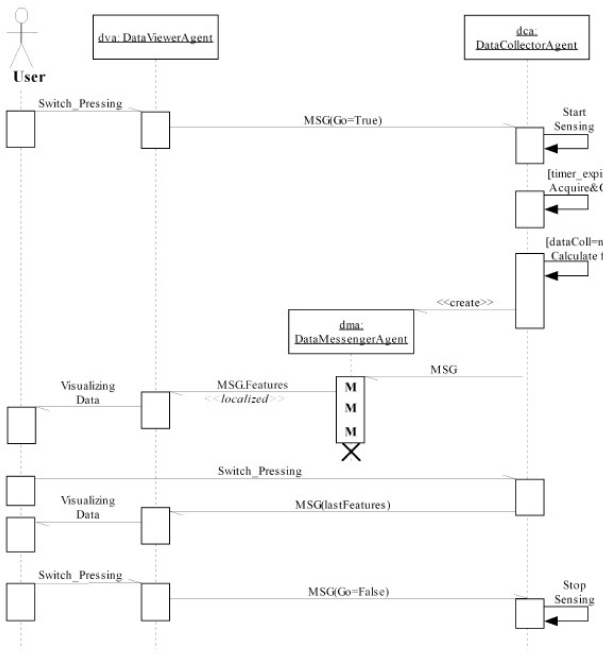

Fig. 7 M-UML sequence diagram for agents interactions

In the following subsections we describe how agent programming models offered by MAPS and AFME can be used to define the DataCollectorAgent behavior.

A. Agent definition in MAPS

As before illustrated, MAPS agents are modeled through a multi-plane state machine. In Fig. 8 the plane related to the DataCollectorAgent is depicted whereas a brief explanation is provided in the following. The *AGN_Start* event causes the transition from the agent creation state to the *IDLE* state whit the execution of an initializing code represented by the action *A0* (e.g. data structures initialization). In the *IDLE* state, when the network message (*MSG*) sent by the DataViewerAgent arrives and the guard *[go==true]* holds, the timer is configure and started for timing the sensors reading (action *A1*) and the agent transits to the WAITSENSING state. When the timer fires (see *TMR_Expired* event), the sensing operations are requested (action *A2*) to the three onboard sensors. When each of the three sensor data are available (see ACC_Tilt, TMP_Current and LGH_current events), their corresponding actions (*A5, A6, A7*) store the values on appropriate buffers. If *numData* samples for each sensor type have not been collected yet, the guard *[dataColl!=numData]* holds so that the agent returns to the WAITSENSING state waiting for the next sensing operations on the timer expiration. If at the contrary the necessary samples number is reached, sensor data are ready for being transmitted to the second sensor node. So, the set of the features are compute and the DataMessengerAgent is created (action *A9*). When the *AGN_Id* event, containing the agent id of the created agent, is received the set of the features values are passed to it (action *A10*). Regarding the WAITSENSING and the DATACOLLECTING states, if the user presses the switch, the last set of computed features are immediately sends to the DataViewerAgent through a remote message (action *A3*). Finally, when the event *MSG* is received and the guard *[go==false]* holds, the agent is terminated (action *A4*).

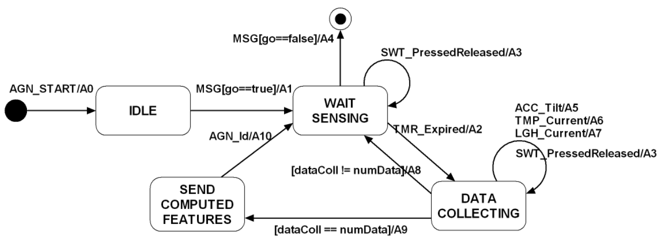

Fig. 8 MAPS-based DataCollectorAgent model

After having described the DataCollectorAgent plane, the code of some actions is provided in Fig. 9. In particular, the most significant part of the Java code related to the plane consists in the operations included into the set of the actions previously defined for the plane state machine.

```
A1:  Event timer = new Event(this.agent.getId(),
        this.agent.getId(), Event.TMR_EXPIRED, Event.NOW);
     timerID = this.agent.setTimer(true, 3000, timer);
A2:  Event temperature = new Event(this.agent.getId(),
        this.agent.getId(), Event.TMP_CURRENT, Event.NOW);
     temperature.setParam(ParamsLabel.TMP_CELSIUS,"true");
     this.agent.sense(temperature);
        //similar code for accelerometer and light
A3:  Event msg = newEvent(this.agent.getId(),
             dataViewerAgentID,Event.MSG, Event.NOW);
     msg.setParam("lastFeatures", this.computedFeatures);
     this.agent.send(this.agent.getId(),dataViewerAgentID,
                                                msg, true);
A6:  this.collectedData += event.getParam(
                ParamsLabel.TMP_TEMPERATURE_VALUE)+"-";
     dataCollTemp++;
A9:     // code for feature computation...
     this.agent.create("applications.demo.Messenger",
        null,this.agent.getMyIEEEAddress().asDottedHex());
A10: Event msg = new Event(this.agent.getId(),
              messengerAgentID, Event.MSG, Event.NOW);
     msg.setParam("features", this.features);
     this.agent.send(this.agent.getId(), messengerAgentID,
                                                msg, true);
```

Fig. 9 Java code for some of the DataCollectorAgent plane actions

B. Agent definition in AFME

The DataCollectorAgent specified trough the AFME design model is depicted in Fig. 10.

The components constituting the previous model are described in the following:

- 7 *Perceptors*. AccPerc, LightPerc and TempPerc are used to acquire data from the SunSPOT sensors. VerifyNumDataSamplesPerc is needed for perceiving if all necessary sensor data have been collected for features computation whereas VerifyFeaturesComputedPerc checks that all features (min, max, mean) have been computed on the collected sensor samples. Finally, TimerPerc checks when the

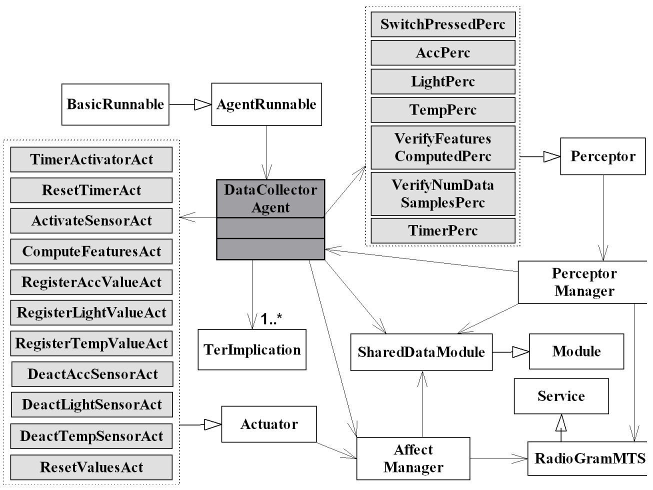

Fig. 10 AFME-based DataCollectorAgent model

timer expires and SwitchPressedPerc recognizes the user switch pressing.

- 11 *Actuators*. TimerActivatorAct and ResetTimerAct are used to activate/reset the timer of the sensor node for timing the sensor acquisition operations. RegisterAccValueAct, RegisterLigthValueAct, and RegisterTempValueAct allow storing the sensed data into the SharedDataModule for sharing them with the proper aforementioned perceptors. DeactAccSensorAct, DeactLigthSensorAct, and DeactTempSensorAct actuators are activated after the sensor readings and are needed to prevent the sensors perceptors from reading sensor data before the timer expiration. In particular, since all the agent perceptors, driven by the AFME runtime system, cyclically perceive the agent environment, and since the sensor data acquisition is made through the use of perceptors, the continuous data acquisition has to be avoided. For this reason, ActivateSensorAct is used to re-enable sensors reading after timer expiration. ComputeFeatureAct is responsible for the actual feature computation on the collected sensor data whereas the ResetValuesAct actuator is used for reinitializing all necessary data structure after a completed feature computation.

- *Rules*. TerImplication contains the auto-generated Java description of the agent behavior rules defined into a script file (see below).

- 1 *Module*. SharedDataModule is the shared memory space which stores the data samples sensed from the node sensors (accelerometer, temperature, light) and other useful data to be shared among perceptors and actuators.

- 1 *Service*. RadiogramMTS represents the transport service for data transmission to remote nodes or basestation.

In the following the basic rules defining the DataCollectorAgent behavior are described:

1. message(inform, sender(dataViewer, addresses("radiogram://” +DataViewerAgentNodeAddr)), ?content), !goFalse > par(timerActivatorAct, adoptBelief(always(goFalse)));
2. timerExpired > par(timerActivatorAct, activateSensorsAct);
3. temperature(?value) > par(deactTempSensorAct, registerTempValueAct(?value));
4. light(?value) > par(deactLightSensorAct, registerLightValueAct(?value));
5. acc(?accX, ?accY, ?accZ) > par(deactAccSensorAct, registerAccValueAct(?accX, ?accY, ?accZ));
6. numDataSampled > computeFeatures
7. featuresComputed(?computedFeatures) > par(resetValuesAct, inform(agentID(messenger, addresses("radiogram://”+ DataMessengerAgentNodeAddr)), values(?computedFeatures)));
8. switch_pressed(?computedFeatures) > inform(agentID(dataViewer, addresses("radiogram://”+ DataViewerAgentNodeAddr)), values(? computedFeatures));
9. message(inform, sender(dataViewer, addresses("radiogram://” +*DataViewerAgentNodeAddr*)), ?content), goFalse > resetTimerAct, deactAccSensorAct, deactLigthSensorAct, deactTempSensorAct;

The rule (1) enables the timer upon the reception of the message coming from the DataViewerAgent and creates the *goFalse* belief. Rule (2) states that after timer expiration the sensor reading operations are activated and also the timer is reactivated. Rules (3, 4, 5) are used for storing the sensor

```
    TempPerc:
....
public void perceive(){
   FOS currentFOS = manager.perManage("shareddata", 5);
   int activated= integer.parseInt(currentFOS.toString()
);
   if(activated == 1) {
      try {
        tempValue= EDemoBoard.getInstance().
                getADCTemperature().getCelsius();
      } catch(IOException e){ e.printStackTrace(); }
      this.adoptBelief("temperature("+ value +")");
   }
}
....
    ActivateSensorAct:
....
public boolean act(FOS arg0) {
   System.out.println("ActivateSensorsAct..");
   m.actOn("shareddata", 2, null);
   return true;
}
....
    Shareddatamodule:
....
   // data structures declaration
   ...
public FOS processPer(int id)throws MalformedLogicExcep{
   ...
   else if(id == 5){
      return FOS.createFOS(""+temperatureSensorActivated);
   }
   ...
}
public FOS processAction(int id, FOS data)
                              throws
                           MalformedLogicExcep{
   ...
   else if(id == 2){
      temperatureSensorActivated= 1;
      lightSensorActivated= 1;
      accSensorActivated= 1;
      return null;
   }
   ...
}
....
```

Fig. 11 Java code excerpt from the AFME DataCollectorAgent

data when are available and at the same time they disable the sensing perceptors. The rule (6) checks if *numData* samples have been acquired for each sensor type, and, in that case, features computation starts. Rule (7) verify that features have been correctly computed and if this is the case, the DataMessengerAgent is notified through a message. Rule (8) states that if the user presses the switch a message with the last computed features is sent to the DataViewerAgent. Finally, the last rule is for checking the reception of a message from the DataViewerAgent and since the *goFalse* belief has been previously inserted by the first rule, the sensing operations are stopped (timer is reset and sensors reading are disabled).

In Fig.11, a code excerpt of the AFME agent is provided and in particular, an example of *perceive* method (from the TempPerc perceptor), *act* method (from the ActivateSensorsAct actuator) and the SharedDataModule are shown.

V. Conclusion

In this paper we have presented several TinyOS-based and Java-based agent platforms for WSNs. In particular, we have described and compared MAPS and AFME in more details, two Java-based platforms offering similar services but through different agent programming models. The MAPS programming model is based on finite state machines and events for defining agent's behavior, whereas AFME provides a BDI-like agent model based on perceptors, actuators and rules. Experimentation experience with MAPS and AFME in the proposed case study and in other application domains suggests that both platforms are effective in supporting agent-based development of WSN application. However, agent programming with AFME is less straightforward than MAPS programming due to the BDI-like AFME agent model, which is more complex than the finite state machine model offered by MAPS. Moreover, finite state machine programming is a model often used in programming embedded systems so it may be more easily exploited by low-level programmers. Ongoing work is devoted to defining a solution for agent communication interoperability between MAPS and AFME agents which would enable the development of heterogeneous agent-based WSN applications.

References

[1] Eiko Yoneki and Jean Bacon, "A survey of Wireless Sensor Network technologies: research trends and middleware's role," Technical Report UCAM-CL-TR-646, University of Cambridge, UK, Sept. 2005.

[2] Danny B. Lange and Mitsuru Oshima, "Seven Good Reasons for Mobile Agents," *Communications of the ACM*, Vol. 42, No. 3 March 1999.

[3] TinyOS, documentation and software, http://www.tinyos.net, (2010).

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

[5] 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 Conf. on Distributed Computing Systems* (ICDCS'05), Columbus (OH), Jun 6-10, pp. 653-662, 2005.

[6] Y Kwon, S. Sundresh, K. Mechitov, 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), pages 1297-1300, 2006.

[7] R. Kent Dybvig, *The Scheme programming language*. Prentice-Hall, 1987.

[8] 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 Multi-agent Systems (AAMAS-09), 12th May, Budapest, Hungary, 2009.

[9] Aiello, F., Fortino, G., Gravina, R. and Guerrieri, A, "A Java-based Agent Platform for Programming Wireless Sensor Networks," *The Computer Journal*. to appear (2010).

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

[11] Muldoon, C., O'Hare, G. M. P., O'Grady, M. J. and Tynan, R., "Agent Migration and Communication in WSNs", in *Proc. of the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies* (2008).

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

[13] 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*, ser. Lecture Notes in Computer Science, vol. 3993. Reading, UK: Springer, 28-31 May 2006, pp. 727–734.

[14] http://www.agentfactory.com (last access June 2010).

[15] A. S. Rao and M. P. Georgeff, "BDI Agents: from theory to practice," *Proceedings of the First International Conference on Multi-Agent Systems* (ICMAS'95), pp. 312–319, June 1995.

[16] Dikaiakos, M., Kyriakou, M., and Samaras, G., "Performance evaluation of mobile-agent middleware: A hierarchical approach," *Proc. of the 5th IEEE Int'l Conference on Mobile Agents*, Atlanta, Georgia, 2–4 December, LNCS 2240, Springer Verlag, Berlin, 2005, pp. 244–259.

[17] Saleh, K., El-Morr, C., "M-UML: An extension to UML for the modeling of mobile agent-based software systems," *Information and Software Technology*, 46(4), pp. 219-227.

[18] Sentilla Developer Community website, http://www.sentilla.com/ developer.html (June 2010).