Integrating Jade and MAPS for the development of Agent-based WSN applications

Mariusz Mesjasz and Domenico Cimadoro and Stefano Galzarano and Maria Ganzha and Giancarlo Fortino and Marcin Paprzycki

Abstract Recent years have seen rapid advancements in wireless sensor networks (WSNs) and software agents resulting, among others, in maturation of their technology platforms. Furthermore, benefits of combining these research areas have been analyzed. The MAPS agent platform allows fusion of agents and WSN's. However, due to the hardware limitation, MAPS misses important functionalities needed, for instance, in advanced decision support. Such functions are available, among others, in the JADE agent platform, geared towards more powerful computing devices. Therefore, integration of MAPS and JADE had to be considered. The aim of this paper is to discuss technical issues involved in achieving this goal.

1 Introduction

Nowadays, wireless sensor networks (WSNs) and (multi-)agent systems (MAS) became popular and fast advancing research areas. Furthermore, it has been suggested that combining WSN's and MAS's can bring about a new generation of intelligent systems [**?**, **?**, **?**]. One of the observable advances in both areas is the rapid maturation of their technology platforms. Currently, there exist four Java-based mobile agent platforms for WSNs: MAPS [**?**, **?**], TinyMAPS [**?**], AFME [**?**] and MASPOT

Mariusz Marek Mesjasz
Warsaw University of Technology, Warsaw, Poland e-mail: `mesjaszm@gmail.com`

Domenico Cimadoro, Stefano Galzarano, Giancarlo Fortino
DEIS – University of Calabria, Rende (CS), Italy e-mail: `g.fortino@unical.it`

Maria Ganzha
Systems Research Institute Polish Academy of Sciences and University of Gdansk, Poland e-mail: `Maria.Ganzha@ibspan.waw.pl`

Marcin Paprzycki
Systems Research Institute Polish Academy of Sciences, Warsaw, Poland e-mail: `Marcin.Paprzycki@ibspan.waw.pl`

[?]. MAPS will be introduced in Section 2 whereas TinyMAPS is a compact and constrained version of MAPS ported on the Sentilla JCreate sensor platform [?]. The AFME framework, is a lightweight version of the AgentFactory framework, ported onto the Sun SPOT [?]. However, AFME was not specifically designed for WSNs and, in particular, for the Sun SPOT environment. MASPOT is a brand new mobile agent system natively designed for the Sun SPOTs. It remains to be seen if its Sourceforge code base will be maintained and updated.

Observe that the hardware capabilities of most sensor networks are restricted by the sensor miniaturization, and the battery life. Therefore, regardless of the WSN-agent platform, agents cannot be expected to perform resource-consuming tasks (e.g. extensive data analysis needed for complex decision support), which require more robust hardware and software. Therefore, in a system supporting a glider pilot (see, [?, ?]) we have proposed a hybrid approach. There, we have decided to proceed with a design of a system, in which MAPS agents are responsible for managing on-board sensors, while JADE agents ([?]) run on a smart device and are responsible for the meta-level "intelligence." For the resulting GliderAgent system we have designed and implemented an initial version of the *gateway* supporting MAPS-JADE integration (inter-communication). The *gateway* was later consolidated and enhanced. The aim of this paper is to discuss, in detail, technical issues involved in its design and implementation.

However, before we proceed, let us note that the mentioned hybrid design can be used in a large number of intelligent systems useful in many application domains such as e-health, smart homes, smart grid, or military scenarios. There, the agent-based WSN subsystem will be dealing with low-level functionalities, e.g. sensor data preprocessing, information routing, or power management. Conversely, the meta-level agent-based subsystem will facilitate data analysis, data mining, or preparation of updated strategies for agents in the WSN subsystem. Hence, the meta-level subsystem will provide the core intelligence of the system.

The remaining parts of the paper are organized as follows. Sections 2 and 3 provide brief introduction of MAPS and JADE agent platforms. The MAPS-JADE Gateway architecture is described in detail in Section 4, whereas preliminary evaluation can be found in Section 5. Finally, concluding remarks, including on-going work, complete the paper.

2 MAPS agent framework

MAPS [?, ?] is a Java-based framework developed for the Sun SPOT technology to enable agent-oriented programming of WSN applications. It has been conceptualized around the following requirements:

- 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, switches and battery).
- Plug-in-based architecture extensions, through which other services can be defined in terms of one or more dynamically installable components implemented as single or cooperating (mobile) agent/s.

The MAPS architecture (see Fig. 1) is based on components that interact through *events* and facilitate message transmission, agent creation, agent cloning, agent migration, timer handling, and access to sensor nodes.

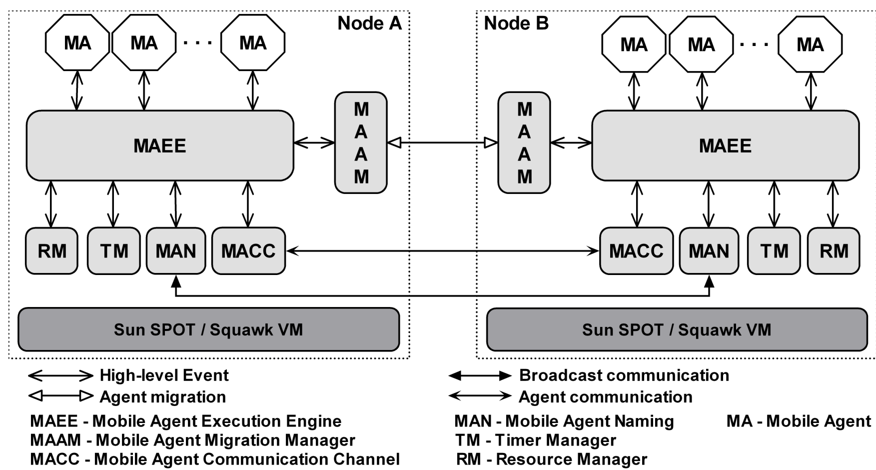

Fig. 1 MAPS architecture – an overview

In particular, the main components of MAPS are:

- *Mobile Agent (MA)* – the basic high-level component, defined by the application programmer.
- *Mobile Agent Execution Engine (MAEE)*, which manages execution of MAs using an event-based scheduler (enabling lightweight concurrency). MAEE interacts with other components, to fulfill service requests issued by MAs (e.g. message transmission, sensor reading, timer setting).
- *Mobile Agent Migration Manager (MAMM)*, which supports agents migration through the Isolate hibernation/dehibernation mechanism provided by the Sun SPOT environment [**?**]. While MAs hibernation and serialization involve data and execution state, the code has to 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)*, enables inter-agent communications based on asynchronous messages supported by the radiogram protocol.

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

3 JADE

The Java Agent DEvelopment framework ([**?**]) is one of the most popular Java-based agent platforms, which complies with the Foundation for the Intelligent Physical Agents ([**?**]) specifications. The *JADE* agent platform is composed of a single *Main Container* and multiple *Agent Containers*, which can be distributed across different hosts. The *Main Container* contains the *Agent Management System* (*AMS*) agent and the *Directory Facilitator* (*DF*) agent. The *AMS* agent supervises the platform, manages the life-cycle of all agents inside it, and provides the *white pages* service (a registry of currently existing agents; including their *Agent Identifiers* (*AID*), used for communication). The *DF* agent provides an optional *yellow pages* service (a registry of services provided by the agents registered in the *AMS*).

All actions undertaken by agents are encapsulated within *Behaviours*, which are executed sequentially in the agent's main thread. However, if an agent has to perform a time-consuming operation (or wait for required resources), *JADE* allows a *ThreadedBehaviour*, which is executed in another thread and does not block the agent. *JADE* agents communicate via *ACLMessages*, which comply with the *FIPA ACL Message Structure Specification* [**?**]. In order to send an *ACLMessage*, an agent has to register an ontology and a codec. The ontology is used to demarcate data inside the *ACLMessage*. The codec encapsulates content, or extracts data from the message (based on the internal structure of the message – message header – and the ontology). The *ACLMessage* is composed of a *header* and one or more *message elements*. The message *header* is always present, and contains information necessary to properly deal with the message (e.g. the source *AID*, the target *AID*, message type, ontology, language (codec), performative, etc.). In *JADE*, message elements can be of a primitive type (e.g., *boolean*, *int*, *string*), or of an aggregate type (any user-defined structure composed of primitive or aggregate elements). Aggregate elements are usually represented as Java classes included in the ontology. Message elements form a content of the *ACLMessage*, which can is represented depending on the codec. For example, a codec can write data in a human readable form (XML, strings), or as a byte code.

4 Gateway Architecture

As stated above, the *JADE/MAPS gateway* (or, simply, the *gateway*) has been implemented to provide a communication mechanism between JADE and MAPS agents. It facilitates bi-directional translation between *JADE ACL messages* and *MAPS Events* and supports routing of communication between the two agent platforms. The *gateway* is composed of two abstract parts – the *JADE part* and the *MAPS part*, responsible for communication with their own platforms. These two parts interact within the *gateway* using *translation mechanisms*. The *JADE part* of the *gateway* is a *JADE agent*. It was a natural choice, because JADE agents can autonomously perform complex tasks. Furthermore, it was established that the translation and routing mechanism should be accomplished with the more powerful environment, thus resulting in assigning this role to a JADE agent that runs on a PC (as the primary target environment).

For the *MAPS part*, there was no simple way to connect the *gateway* to the MAPS platform. Placing any part of the *gateway* on a Sun SPOT device could result in a communication bottleneck due to its limited resources. Since it is impossible to run MAPS agents without the MAPS Execution Engine, to allow its execution outside of the Sun SPOT devices (e.g. on a PC), the MAPS platform would have to be rewritten. However, implementation of a fully functional PC-based MAPS platform would not help solving the cross-platform communication problem. Instead, it was enough to reimplement selected parts of the MAPS Execution Engine that (i) are responsible for communication over the radio, and (ii) manage the list of remote MAPS agents. Hence, the *gateway* was developed as a *semi-functional MAPS Execution Engine* without the capability of running actual MAPS agents.

Let us now describe the *JADE-MAPS communication* from three perspectives: (1) communication within JADE, (2) communication within MAPS, and (3) passing information between the two platforms.

Recall that the *gateway* is a *JADE agent*. Therefore, it can be directly accessed by other JADE agents via the *AMS* and *DF* services (see section 3), and can communicate with them using ACL messages. However, to facilitate JADE-MAPS communication, the *gateway* provides a special ontology (*GatewayOntology*), which describes actions that need to be performed by the *gateway*: *Register*, *Unregister*, *GetRemoteAgent* and *SendMessage*. Each action is represented by a Java class, which is used to fill the content of an *ACLMessage*. The ACL message containing one of these actions has the performative set to *REQUEST*. The *gateway* can respond to such messages with an *INFORM* message, containing a confirmation of execution of the requested action (and a list of MAPS agents, in the case of *GetRemoteAgents*), or with a *FAILURE* message containing a string, specifying the cause of the failure (see Fig. 2). If the JADE agent requests communication by sending the *SendMessage* action, the *gateway* may optionally send an additional *INFORM* message with a response from a MAPS agent (an instance of the *ReceivedMessage* class, also included in the *GatewayOntology*).

A JADE agent, which wants to communicate with MAPS agents, has to register itself within the *gateway*. To do this, the agent has to send an ACL message spec-

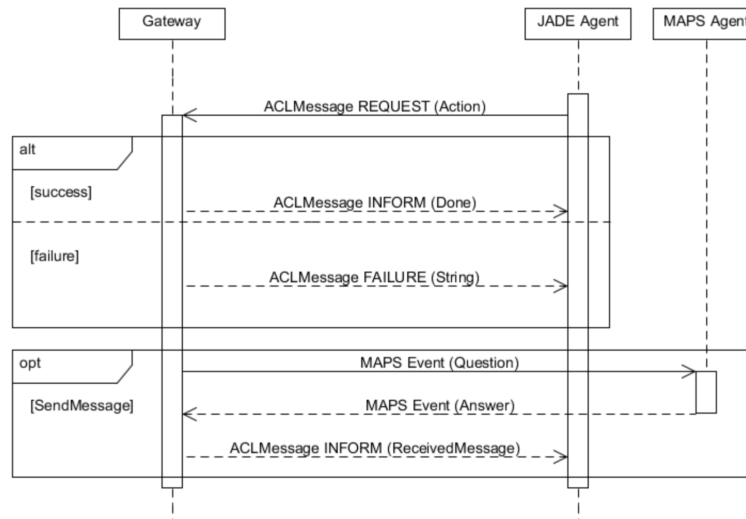

Fig. 2 A sequence diagram which represents a typical communication with the *gateway*

ifying the *Register* action. When the *gateway* receives such a message, it creates a unique *MAPS ID* (for the sender's *AID*) and sends it within a *MAPS REFRESH* message to the *MAPS Execution Engines*, to allow agent synchronization across the platform. Next, the *gateway* adds a pair (*AID*, *MAPS ID*) to its list of local agents. These agent identifiers are used during the message translation process. All registered *JADE agents* should unregister themselves at the end of their life-cycle by sending an ACL message with the *Unregister* action. In response, the *gateway* removes the pair (*AID*, *MAPS ID*) from the memory and sends a *MAPS REFRESH* message to the *MAPS Execution Engines*, informing that a given *MAPS ID* is no longer valid.

From the MAPS perspective, the *gateway* is just another *MAPS Execution Engine*. To facilitate this, the *gateway* broadcasts a *MAPS PUBLISH* message, which (1) makes the *gateway* available within the MAPS platform, (2) discovers all available Sun SPOT devices that run MAPS, and (3) initializes the lists of MAPS agents. This list is later updated, based on the communication between the *gateway* and the other MAPS Execution Engines.

The JADE agents are accessible to the MAPS agents by their *MAPS IDs*, which correspond to their *AIDs*. The MAPS agents are *not* aware that the *gateway* (and the JADE agents) belong to a different platform. Hence, communication between the MAPS and the JADE agents is simplified to communication between MAPS agents.

The translation mechanism "combines" the two platforms and is invoked when the *gateway* receives a message, which has to be translated either into an ACL message, or into a MAPS Event. The translation begins with the creation of an appropriate message header. Since an ACL message and a MAPS Event are very different, the *gateway* had to introduce a common communication standard. As presented in

```
ACL Message - question

(REQUEST
 :receiver (set (agent-identifier : name 123456-MAPS@192.168.1.2:1099/JADE
                 :addresses (sequence http://Mariusz-VAIO:7778/acc )) )
 :content "((action (agent-identifier :name 123456-MAPS)
    (Send
        (Message
            :msgName 41
            :msgType 1
            :parameters (sequence data temperature)))))"
 :language fipa-sl :ontology JADE-MAPS-gateway-ontology )

MAPS Event - question

0000.9876GLIDER$0000.123456$41$1$data&temperature

MAPS Event - answer

0000.123456$0000.9876GLIDER$41$1$data&36.99999999

ACL Message - answer

(REQUEST
 :receiver (set (agent-identifier : name GliderAgent@192.168.1.2:1099/JADE
                 :addresses (sequence http://Mariusz-VAIO:7778/acc )) )
 :content "((action (agent-identifier :name GliderAgent)
    (Message-for-you
        (Message
            :msgName 41
            :msgType 1
            :parameters (sequence data 36.99999999)))))"
 :language fipa-sl :ontology JADE-MAPS-gateway-ontology )
```

Fig. 3 An example of the simple translation mechanism. The *ACLMessages* has been encoded by the *SLCodec*.

Fig. 3, the ACL message contains the message header and a *SendMessage* action, which not only requests to send a message, but also provides additional information encapsulated within the *Message* concept. The *Message* concept includes a source *AID*, a target *MAPS ID*, a message name (here, topic 41 means that this is an ordinary message), a message type (occurrence time 1, means "now") and parameters. Obviously, this information is MAPS-specific and does not have an equivalent in the ACL message (for example, the message name *TEMPERATURE* does not match any property in the ACL message). A corresponding *MAPS Event* header is created, based on this information. The only field that requires translation is the agent *AID* (valid only inside the JADE platform). Here, the *gateway* translates the *AID* based on the list of (*AID*, *MAPS ID*) pairs (e.g. in Fig. 3, the *gateway* found the pair (*Peter@192.168.1.3:1099/JADE*, *0000.98765XYZW*) and used it in the corresponding MAPS Event).

The translation from a MAPS Event to an ACL message is the reverse process. The *gateway* starts with an empty *ACLMessage* class of the type *INFORM*. The *AID*, corresponding to the target *MAPS ID* (from the MAPS Event), is set as the receiver of this message. However, the *gateway* has to specify the context of the MAPS Event (the message name and the message type). Since the ACL message header does not contain the MAPS-specific information, the *GatewayOntology* provides a predicate called *ReceivedMessage* (denoted by the string "Message-for-you"). The predicate contains only one field, which stores the *Message* concept. This *Message* concept

is filled with MAPS information by the *gateway*. Notice that the ACL messages, presented in Fig. 3, differ from each other to a small extent. Namely, the ACL message type changes from *REQUEST* to *INFORM* and, respectively, the content of the message is changed from the *SendMessage* action to the *ReceivedMessage* predicate. Otherwise, the translation mechanism does not alter the internal structure of MAPS Events.

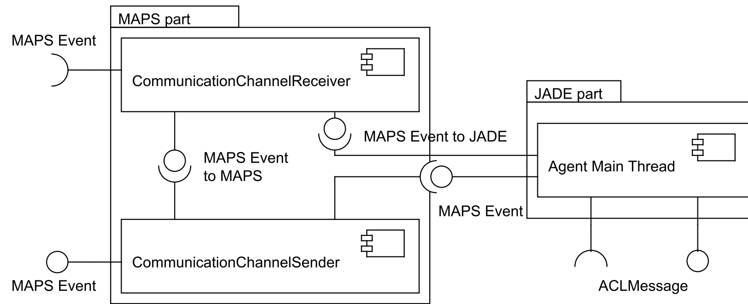

Fig. 4 A component diagram of the gateway.

Technically, the *gateway* uses three threads: (i) *CommunicationChannelReceiver* (*CCR*), (ii) *CommunicationChannelSender* (*CCS*), and (iii) the *Agent Main Thread* (see Fig. 4). The *CCR* and the *CCS* originate from the *MAPS Execution Engine* source code, and are used inside the *gateway* as separate *ThreadedBehaviour*s. The *CCR* has to be run as a separate thread due to the asynchronous communication within the *MAPS platform*. The execution of the CCS as a different thread guarantees the absence of communication bottleneck, as discovered during tests on an earlier version of the gateway (see [**?**]).

Each time the *CCS* receives a MAPS Event, the datagram must be checked for the destination address. If the MAPS Event is addressed to another MAPS agent or an Execution Engine within the MAPS platform, it is forwarded directly. However, if the MAPS Event has to be delivered to a JADE agent (with a *MAPS ID* available in the *gateway*), then it is sent to the *Agent Main Thread* for further processing.

For each message, the agent main thread has to determine its destination. If this is a local JADE-to-JADE message, it is processed inside the *JADE part* of the *gateway*. If the message has to be sent to a MAPS agent, then the translation mechanism is invoked and the corresponding *MAPS Event* is put in the *CCR* queue, to be forwarded to an appropriate MAPS Execution Engine. Note that the *gateway* introduces two restrictions: (a) two JADE agents inside the same gateway cannot communicate with each other via MAPS Events, and (b) there is no confirmation that a MAPS agent will receive, or respond to, an ACL message. The first restriction is to reduce the workload of the *gateway*. The second restriction is due to lack of a mechanism to monitor message traffic in the radio network used by the MAPS platform.

5 Preliminary performance test

To check the performance of the *gateway*, we executed a communication test between a number of agent pairs. Each pair was composed of one JADE and one MAPS agent. This test was designed to flood the system with ACL messages and MAPS Events. During the test, we incremented the number of agent pairs from 1 to 20. Frequency of message transmission was set to 500 milliseconds, and the message size was 8 bytes (representing the standard size of the value of temperature) plus the size of the message header. The *Forwarding Time* (*FT*) was measured, and represents the time needed by the gateway to forward a message from the Jade agent, plus the time needed to receive a reply form the MAPS agent ("round-trip" communication). Specifically, during each experiment, a sender (*JADE agent*) sends a message to a receiver (*MAPS agent*) to request the value of the temperature, and receives a response. Each test lasted 5 minutes to complete, and a total of 20 experiments have been carried out to obtain a good confidence measure.

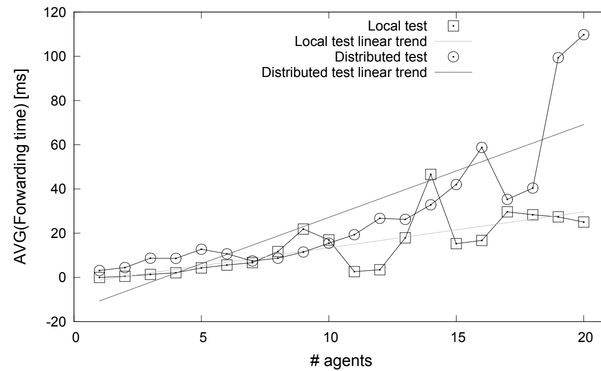

Fig. 5 Forwarding time for multiple agents.

The experimental results are presented in Fig. 5. The *FTs* for the distributed test (agents on different machines) are greater than the ones obtained on a single host (due to the use of the RMI protocol to communicate between JADE agents and the *gateway*). This is the reason why the linear trend corresponding to the local test grows faster and its slope is steeper. Moreover, the increase in the number of agents results in a greater variation of the *FT*. Evaluation shows also the performance degradation due to the time-consuming operations (serialization and radio stream-based communication) performed by the *Sun SPOT libraries*. Hence, the performance of the *gateway* is highly influenced by the Sun SPOT emulator (the *Solarium*).

6 Concluding remarks

In this paper we have proposed and described a *gateway* component providing inter-communication capabilities between MAPS and JADE agent platforms. The importance of such gateway is in providing better support for the development of intelligent WSN systems, where the small footprint MAPS agents facilitate sensor management, whereas JADE agents infuse the system with intelligence. In the near future, we plan to carry out more complete performance evaluations of the *gateway*, and to include it in the JADE plug-in repository.

References

1. Fipa acl message structure specification. `http://www.fipa.org/specs/fipa00061/SC00061G.html`
2. The foundation for intelligent physical agents (FIPA). `http://www.fipa.org/` (2010.03.13)
3. Mobile Agent Platform for Sun SPOT (MAPS), documentation and software. `http://maps.deis.unical.it` (2011)
4. Sentilla developer community. `http://www.sentilla.com/` (2011)
5. Sun Small Programmable Object Technology (Sun SPOT), documentation and software. `http://www.sunspotworld.com` (2012)
6. Aiello, F., Fortino, G., Galzarano, S., Gravina, R., Guerrieri, A.: An analysis of java-based mobile agent platforms for wireless sensor networks. Multiagent and Grid Systems **7**(6), 243–267 (2011)
7. Aiello, F., Fortino, G., Galzarano, S., Vittorioso, A.: TinyMAPS: a lightweight Java-based Mobile Agent System for Wireless Sensor Networks. In: Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC2011), *Studies in Computational Intelligence*, vol. 382, pp. 161–170. Springer-Verlag, Berlin, Heidelberg (2011)
8. Aiello, F., Fortino, G., Gravina, R., Guerrieri, A.: A java-based agent platform for programming wireless sensor networks. The Computer Journal **54**(3), 439–454 (2011)
9. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with a fipa-compliant agent framework. Softw., Pract. Exper. **31**(2), 103–128 (2001)
10. Domanski, J.J., Dziadkiewicz, R., Ganzha, M., Gab, A., Mesjasz, M.M., Paprzycki, M.: Implementing glideragent—an agent-based decision support system for glider pilots. In: Software Agents, Agent Systems and Their Applications, pp. 222–244 (2012)
11. Gab, A., Adreout, P., Ganzha, M., Paprzycki, M.: Glideragent-a proposal for an agent-based glider pilot support system. In: Proceedings of the 15th International Conference on Methods and Models in Automation and Robotics (MMAR), pp. 55–60. IEEE Press (2010)
12. Lopes, R., Assis, F., Montez, C.: MASPOT: A Mobile Agent System for Sun SPOT. In: Proceedings of the 2011 Tenth International Symposium on Autonomous Decentralized Systems, ISADS ’11, pp. 25–31. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/ISADS.2011.10
13. Muldoon, C., O’Hare, G., O’Grady, M., Tynan, R.: Agent migration and communication in WSNs. In: 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 425–430. IEEE (2008)
14. Rogers, A., Corkill, D.D., Jennings, N.R.: Agent technologies for sensor networks. IEEE Intelligent Systems **24**, 13–17 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/MIS.2009.22
15. Vinyals, M., Rodríguez-Aguilar, J.A., Cerquides, J.: A survey on sensor networks from a multi-agent perspective. The Computer Journal **54**(3), 455–470 (2010)