USEME: A Service-oriented Framework for Wireless Sensor and Actor Networks *

Eduardo Cañete, Jaime Chen, Manuel Díaz, Luis Llopis and Bartolomé Rubio
Dpto. Lenguajes y Ciencias de la Computación. Málaga University
29071 Málaga, SPAIN
(ecc,hfc,mdr,luisll,tolo)@lcc.uma.es

Abstract

We are in the presence of a new and powerful technology called Wireless Sensor and Actor Networks. There are many fields where we can apply this technology to develop varied and interesting applications: high security environments, environmental monitoring, industrial monitoring, medicine, precision agriculture. This technology brings the need to develop new frameworks in order to make easier the application developer's task. Recently, different high-level programming abstractions and middleware have appeared as promising solutions. In this paper, a new service-oriented framework is introduced. The general scheme of the framework and a detailed description of the programming model are presented. The approach is oriented to deploy lightweight services on sensors and actors. Services can be composed among them by means of the port concept to form complex ad-hoc systems. A building monitoring and control application is described as a motivation example and it is used along the paper in order to show the expressiveness and usability of the abstract programming language proposed.

1. Introduction

Wireless Sensor and Actor Networks (WSANs) constitute a new pervasive and ubiquitous technology and currently on of the most interesting fields of research. Due to a combination of recent technological advances in electronics, nanotechnology, wireless communications, computing, networking, and robotics, it is now possible to design advanced sensors (tiny, low-cost and low-power nodes, colloquially referred to as "motes") that can be deployed in the environment in order to gather information about physical phenomena and report it to actor devices which are able to react by affecting the environment in order to tackle the problem [1]. WSANs offer numerous advantages over traditional systems, such as the large-scale flexible architecture (potentially hundreds or thousands of motes), high-resolution sensed data and application adaptive mechanisms. These unique characteristics make these systems very useful for a wide range of application areas [5].

WSAN applications are inherently difficult to develop and deploy. Programming this kind of system has traditionally been an error-prone task since it requires programming individual nodes, using low-level programming issues and interfacing with the hardware and the network. Furthermore, WSANs have two major requirements: coordination mechanisms for both sensor-actor and actor-actor interactions, and real-time communication to perform correct and timely actions. The complexity of designing and implementing this kind of application makes the supply of higher-level abstractions of low-level functionality necessary in order to ease the application programmer task. In the last few years, different high-level programming abstractions and middleware have appeared as promising solutions to address the challenges of this kind of system. Two interesting surveys classifying them can be found in [6] [10].

Recently, the Service Oriented Architecture (SOA) has been considered as a good candidate to develop open, efficient, inter-operable and scalable WSAN applications [9]. In Service Oriented WSANs (SO-WSANs), node's sensing and actuation capabilities are exposed in the form of in-network services. Application development is simplified by providing standards for data representation, service interface description, and service discovery facilitation. By wrapping application functionality into a set of modular services, a programmer can then specify execution flow by simply connecting the appropriate services together. Some approaches are TinySOA [9], OASiS [7] and TinyWS [8]. In TinySOA, services are lightweight code units deployed directly on top of the operating system of nodes. Applications invoke services using a service-oriented query model. Queries are submitted to one of the established base stations

*This work is supported by the EU funded project FP6 IST-5-033563 and the Spanish project TIC-03085

or directly to individual nodes. OASiS also uses a passive discovery mechanism, but it is combined with an object migration approach instead of using remote query mechanisms. Finally, TinyWS is a small web service platform that resides on the sensor nodes. It hosts the web services and has SOAP processing engine. The sensor nodes are service providers, the application devices are service requestors and a distributed UDDI acts as an overlay entity.

We introduce USEME, Ubiquitous SErvices on Mote Environments, a new framework that uses a service-oriented high-level programming model and the corresponding middleware support to develop WSAN applications. In this paper the general scheme of the framework and a detailed description of the programming model are presented. Instead of using query models, object migration or web service technology, our approach provides the application programmer with an abstract programming language in order to specify, in a declarative way, the services that the different network nodes (sensors and actors) provide or require. By means of ports, which define commands and events, services interact with each other and can be composed to form complex ad-hoc systems. Group formation are considered in order to achieve more scalable systems. Priority, period and deadline issues are taken into account in order to deal with real-time requirements at the service specification level. The proposal will abstract developers from low-level implementation tasks, such as discovery, communication, group formation or real-time issues. The abstract programming language is platform independent and different approaches can be considered to achieve service implementations.

The rest of the paper is structured as follows. In Section 2 a building monitoring and control application is shown as a motivation example. Section 3 presents the general scheme of the proposed framework. The abstract programming language is described in Section 4. Section 5 describes some implementation issues that are being considered in our tentative prototypes. Finally, some conclusions are sketched in Section 6.

2. Motivation

Building monitoring and control can be considered as a typical application of WSANs. Figure 1 shows the different services the system is normally required to perform:

- *Indoor environmental monitoring (heating, ventilation and air conditioning services).* An air conditioner acts as an actor device that periodically receives measurements from temperature and humidity sensors. The air conditioner adjusts the air quality to meet user preferences.

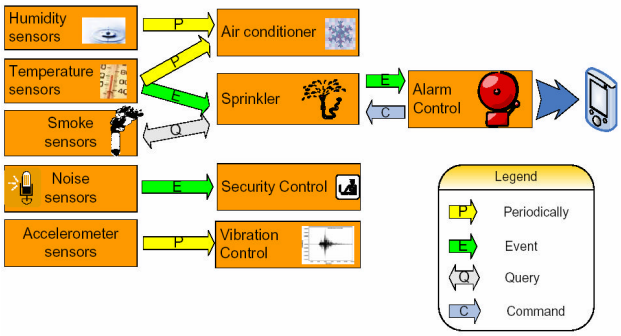

Figure 1. Schematic scenario of services in a building monitoring and control application.

- *Response to extreme events such as fire.* When a water sprinkler device sporadically receives a high-temperature event, it asks for the smoke level. If smoke detectors actually report the presence of fire, the water sprinkler is operated and an event is sent to a fire alarm control device, which can then activate emergency bells, send an emergency message to some subscribed target (e.g. the PDA of a firefighter) and activate some other nearby water sprinklers.

- *Structural monitoring.* A vibration control device periodically receives measurements from accelerometer sensors so that possible vibrations suffered by the building can be controlled.

- *Security control.* Sporadic noise level events can be sent from the corresponding sensors to actor devices able to activate cameras and send messages to centralized control devices.

Each of the aforementioned goals involves only a specific part of the system. Keeping the processing close to where data is sensed has been long recognized as an effective approach to save energy, achieve more efficient implementations and support real-time requirements [1]. In applications following a sense-to-react pattern it is indeed unreasonable to send the sensed data to a single, powerful base station, as this may negatively affect latency and reliability. Therefore, it is better to organize network nodes in different clusters or groups. For example, different temperature and humidity sensors can be deployed in a room forming a group together with an air conditioner. In order to carry out its heating, ventilation and air conditioning services, the air conditioner needs temperature and humidity services provided by the corresponding sensors. In the same way different groups can be formed in the building in order to achieve the required activities.

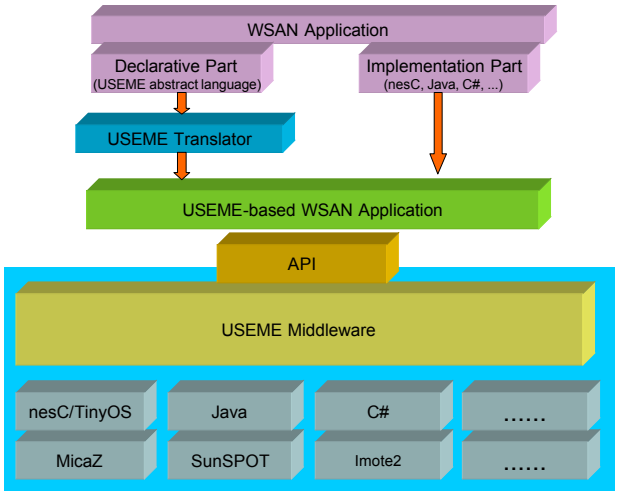

Figure 2. Framework Scheme.

3. The USEME Framework

Figure 2 depicts the general scheme of the USEME framework. The application programmer has to elaborate two different parts in order to obtain a USEME-based WSAN application:

- A declarative part where the abstract programming language described in the following section is used to specify definitions of the main elements involved in the system: nodes (sensors and actors), groups and services, as they were identified in the motivation example. This part is common to every node in the network.

- An implementation part that contains a node creation and the implementation of the different services that it provides. In addition, group creations can also be established. Node and group creation is carried out using the abstract programming language proposed. On the other hand, service implementation is USEME-independent. For example, the application programmer can use the component model provided by the nesC language [4] in order to be used on MicaZ-based motes [3] or object oriented languages such as Java, C++ or C# to be used on actor devices or new generation motes such as SunSPOT [12] and Imote2 [3].

Following the terminology established in the literature [2], our approach combines macro-programming with (enhanced) node-centric programming. Macro-programming of sensor networks broadly refers to an application development methodology that focusses on the desired global behavior liberating the programmer from having to compose the complex control, coordination and state maintenance mechanisms at the individual node. In node-centric programming, the programmer has to translate the global application behavior in terms of local actions on each node, and individually program the sensor nodes using the corresponding language, interfacing local sensing interfaces and sending/receiving messages from other nodes.

In our approach, the declarative part of an application establishes the WSAN global behavior at a high-level programming abstraction using the service-oriented paradigm. Section 4 will describe the abstract language used to establish this service composition-based global behavior. On the other hand, the implementation part follows a node-centric programming style as the services provided by a particular node have to be programmed. However, the use of a service-oriented approach together with the logical way the groups are tackled in our proposal increase the level of programming abstraction. The middleware support provides facilities to address the different nodes in terms of logical, dynamic relationships among the services involved in the application.

The declarative part constitutes the input to a translator whose output together with the implementation part form the final USEME-based WSAN application on each node. This application uses the different operations offered by the middleware API in order to create a group, publish a service, call a port command, receive an event, etc.

4. The USEME Abstract language

In the USEME framework, applications are designed as a composition of services provided by the different WSAN nodes. The service-oriented high-level programming model supported is based on the use of three major elements: network nodes, i.e. sensors and actors, services published by these nodes and the concept of group as a main issue to achieve scalability and efficiency. Figure 3 shows an example graphical view where these elements appear. A node can publish its services in the context of one or more groups. For example, node `E` publishes its services `SE1` and `SE2` in group `G2`. On the other hand, node `C` publishes its service `SC` in groups `G1` and `G2`. Services published in a group can only be accessed by members of this group. Therefore, the group is a key concept for service composition.

Service composition is carried out by means of ports. A port defines commands implemented and events raised by the service that provides it. On the other hand, another service requiring this port can call the commands and receive the events defined. For example, service `SB` published by node `B` provides a port that is required by service `SA` published by node `A`. Therefore, service `SA` can call any command defined in the port and can receive any event raised by service `SB`.

This generic example depicted in Figure 3 could repre-

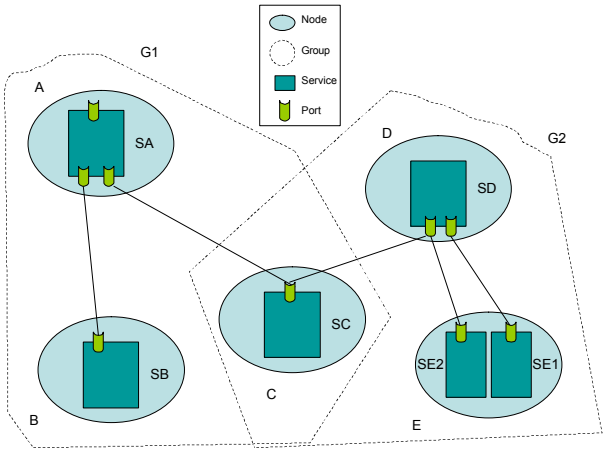

Figure 3. Service Composition Scheme.

```
// Node template
<node_template> ::= [Sensor | Actor] template <node_templ_name>
          ['(' <par_name> {, <par_name>}')']
          '{' <attribute_bindings> <service_publishings> '}'
<attribute_bindings> ::= {<attribute_name> = <attribute_values>;}
<attribute_values> ::= (<par_name> | <value>)
                    {(and | or) (<par_name> | <value>)}
<service_publishings> ::= {Publish <services> in groups <groups>;}
<services> ::= <service_name> {, <service_name>}
<groups> ::= <group_templ_name> ['(' <value> {, <value>}')']
          {,<group_templ_name> ['(' <value> {, <value>}')']}
<value> ::= <string> | <integer>
// Node creation
<node_instance> ::= Create (Sensor | Actor) <node_templ_name>
                    ['(' <value> {, <value>}')'];
// Group template
<group_template> ::= Group template <group_templ_name>
          ['(' <par_name> {, <par_name>}')']
          '{' <group_members> <cardinality> <battery_level> '}'
<group_members> ::= Devices = (Actor | Sensor | Both); <attr_predicates>
<attr_predicates> ::= {<attribute_name> = <attribute_values>
              on (Actor | Sensor | Both);}
<cardinality> ::=
     [Cardinality = {'(' (Sensor | Actor | <par_name> | <string>) ,
          <range_value> - <range_value>')'};]
<range_value> ::= <integer> | <par_name>
<battery_level> ::= [MinBatteryLevel = <range_value>;]
// Group creation
<group_instance> ::= Create Group <group_templ_name>
                    ['(' <value> {, <value>}')'];
// Services
<service_definition> ::= Service <service_name>
          '{' <description> <ports> '}'
<description> ::= Description = <string>;
<ports> ::= {(Provides | Requires | Optional) <port_name> [<constraint];}
<constraint> ::= with constraints '{' <constraints> '}'
<constraints> ::=
     {on <cmd_or_ev_name> <constraint_type> {, <constraint_type>};}
<constraint_type> ::= <real_time_constraint> | <group_constraint>
<real_time_constraint> ::= (Deadline | Period | Priority) <integer>
<group_constraint> ::= Average | Min | Max | NodeId
// Ports
<port_definition> ::= Port <port_name> '{' <commands_and_events> '}'
<commands_and_events> ::= {(CommandSync | CommandAsync | Event)
               <name> ['(' <arg> {, <arg>}')'];}
<arg> ::= (in | out) <arg_type> <arg_name>
```

Figure 4. BNF Syntax for the USEME abstract language.

sent a part of our motivation application described in Section 2. Groups `G1` and `G2` could identify the "water sprinkling" and "air conditioning" groups deployed in a building room, respectively. This way, node `A` is the actor device providing the water sprinkler activity (service `SA`), which requires both temperature and smoke services (services `SC` and `SB`, respectively). On the other hand, node `D` carries out the control of the air conditioner (service `SD`), which requires both temperature and humidity services. Node `E` is a sensor type supplying these two services (`SE1` and `SE2`). In addition, service `SC` can also be used as it supplies temperature values.

The following sections offer a detailed explanation of the abstract programming language used to support our service-oriented model. Figure 4 summarizes the BNF syntax of the language.

4.1. Nodes

In our abstract language, a node type is defined by means of a node template. Several parameters can be used in order to fix different node attributes. Besides attribute bindings, services provided by the node type and the groups where they are published are specified. After node type definition, different nodes can be instantiated by means of the `Create` construct. Figure 5 shows an example where a node template is defined for a sensor node that has three attributes: its identifier, the location where it will be deployed and the variety of sensing modalities. The first two attributes are bound to the values received as parameters. The third one indicates that the node is able to sense temperature, light and sound. The temperature service provided by the node is published in both the "air conditioning" (`AirCond`) and the "water sprinkling" (`WaterSpr`) groups formed in the location established by the parameter. Finally, a sensor is created in the "room245" and with the identifier 1.

4.2. Groups

As said before, the group is a key concept for service composition and is a main issue to achieve scalability and efficiency. It adds a new level of abstraction to join nodes with common restrictions. Our approach allows us to establish certain group constraints by means of a group template:

- *Group members*. By means of the `Devices` construct, the member type (`Actor`, `Sensor` or `Both`) of a group is established.
- *Attribute constraints*. Optional predicates on node attributes can be established. They have to be checked against actual node instances to determine whether they belong to the group.

```
Sensor template MTSSensor(ident, loc)
{
    Identifier = ident;
    Location = loc;
    SensorType = "Temperature" and "Light" and "Sound";
    Publish TempService in groups AirCond(loc), WaterSpr(loc);
}

Create Sensor MTSSensor(1, "room245");
```

Figure 5. Sensor node definition and instantiation.

```
Group template WaterSpr(loc)
{
    Devices = Both;
    Location = loc;
    SensorType = "Temperature" or "Smoke" on Sensor;
    Cardinality = (Actor, 1-1)
                  ("Temperature", 1-3) ("Smoke",1-2);
}

Create Group WaterSpr("room245");
```

Figure 6. Water Sprinkling group definition and creation.

- *Cardinality*. We can specify how many sensors and actors must belong to a group.
- *MinBatteryLevel*. It establishes the minimum level of battery required for a node to belong to the group.

Figure 6 shows an example where a "water sprinkling" group is defined as a group formed by several sensors and only one actor, all of them in the same location (given as a parameter). A sensor must have the value `Temperature` or `Smoke` bounded to its `SensorType` attribute. The number of temperature sensors required ranges from 1 to 3, and for smoke sensors this range is 1-2. Finally, a group is created in the "room245". Each time a node tries to publish its services in the group, the established group constraints are checked in order to determine if the node belong to the group.

4.3. Services

A service definition consists of a service description followed by the ports that will take part in the service in order to allow it to be composed with other services. Service ports are classified into three categories:

- *Provided ports*. They are the ports that the service offers to other services.
- *Required ports*. They are the ports that the service needs from other services in order to be executed.

```
Port TempPort
{
    CommandSync GetTemp(out int value);
    CommandAsync SetThreshold(in int value);
    Event HighTemp(out int value);
    Event Temp(out int value);
}

Service TempService
{
    Description = "Temperature Service";
    Provides TempPort;
}
```

Figure 7. Temperature service definition.

- *Optional ports*. They are required ports associated with non crucial operations so that the service can achieve its function despite they are not available.

A port definition can include commands (synchronous and asynchronous) and events (which are asynchronous communication mechanisms). As said before, commands are implemented and events raised by the service that provides the port where they are defined. Figure 7 shows the definition of a temperature service (`TempService`), which provides a port (`TempPort`). This port defines two commands and two events. `GetTemp()` is a synchronous command that can be used by a service at any time to get the temperature value sensed. The `HighTemp()` event is raised when the temperature sensed is higher than a established threshold. This value can be modified by means of the asynchronous command `SetThreshold()`. Both `HighTemp()` and `SetThreshold()` can be used by the water sprinkler service as established in our motivation example. Finally, `Temp()` is a periodical event providing the temperature sensed and it can be used by our air conditioner service.

Our approach allows the developer to establish real-time constraints for a command or event when a service is being defined. These constraints can be specified by means of the following keywords:

- *Priority*. One of the characteristics that contributes to achieving the real-time requirements demanded in WSANs is the priority issue. Some activities are more important than others and should be scheduled in an appropriate way in order to enhance the system response time. In our approach, it allows us to establish the priority of both commands and events of a port provided by the service.
- *Deadline*. It establishes the maximum execution time expected for a command and it is specified in the required (and optional) ports.
- *Period*. It defines the time interval for a periodical event. It can be establish in both provided and required

```
Service TempService
{
    Description = "Temperature Service";
    Provides TempPort with constraints {
        on GetTemp Priority 3;
        on HighTemp Priority 1;
        on Temp Priority 2, Period 60000;
    };
}

Service AirCondService
{
    Description = "Air Conditioner Service";
    Requires HumidityPort;
    Requires TempPort with constraints {
        on Temp Period 1200000;
    };
}
```

Figure 8. Real time constraints.

(and optional) ports. The run-time system must analyze the possible differences between a required and a provided period for an event in order to assess the possibility of matching.

Figure 8 shows the definition of a temperature service (`TempService`), which provides a port (`TempPort`) with some real-time constraints. `HighTemp()` has the higher priority. `SetThreshold()` has the lower one (as it is not specified). In addition, the period of `Temp()` is established to 1 minute. On the other hand, in the definition of the air conditioner service (`AirCondService`) the `Temp()` event is required every 2 minutes. In this case, the run time system will be able to adapt the event reception to the required timing constraints.

Finally, other interesting kind of constraint affecting service composition can be specified. They refer to the number of nodes involved in the provision of a command and the way the required data are tackled. By default, when a service invokes a required synchronous command it is applied to any node providing it in the group. On the other hand, if the command is asynchronous, it is applied to every node providing it in the group. This default behavior can be changed using the following keywords when requiring a command of a port:

- *NodeId*. It allows us to specify a particular node, which will be carried out in the implementation part.
- *Average, Max, Min*. The command will provide the average, maximum or minimum value of all nodes providing it in the group.

5. Implementation Plan

Our service-oriented high-level programming model, and more specifically the abstract language supporting it, is independent from the underlying hardware platform and operating system. However, to verify the feasibility of our approach and evaluate its performance on a real system, we are currently developing three prototypes using three different mote technologies.

The first prototype is based on the use of the well-known and broadly used Crossbow family MicaZ motes. These devices have a 8 MHz Atmel ATMega128L 8-bit microprocessor connected to a 2.4 GHz Chipcon CC2420 (IEEE 802.15.4 compliant) radio transceiver. The platform has available 4 KB of RAM and a flash memory with 640 KB (128 KB for program and 512 KB for user data). Different sensor boards with a variety of sensing modalities can be connected. MicaZ motes run TinyOS [13], a simple but highly concurrent open source operating system, which has been implemented using nesC, a C-based programming language for networked embedded systems, such as sensor networks. The constraints of this kind of system represent a great challenge for us in order to be able to implement our approach principles in an efficient way.

The other two prototypes are thought to use new generation and more powerful motes: Imote2 and SunSPOT. On the one hand, the Imote2 motes (also from Crossbow) have an Intel PXA271 32-bit XScale processor at 13–416 MHz, 256 KB of SRAM, 32 MB of SDRAM and 32 MB of flash memory. We are using several Imote2.Builder kits, where Imote2 modules are pre-programmed with the Microsoft .NET Micro Framework. On the other hand, SunSPOT has a 180MHz 32-bit ARM920T core processor with 512KB RAM and 4MB Flash. These devices implement a full Java Platform, Micro Edition (Java ME) Virtual Machine. Both devices have the same radio transceiver as MicaZ motes. With these prototypes we will try to analyze and evaluate the new possibilities that these promising systems can offer in order to use high-level abstraction solutions and more specifically service-oriented ones to the development of WSAN applications.

For each prototype, two core components must be provided for our approach to become available to the programmer. The first one is the language support. Our plan is to develop a translator that taking as input the declarative part of an application (USEME abstract language) and the implementation part with the corresponding service implementations (nesC, Java or C#), generates a (nesC, Java or C#) program that uses the operations supplied by the middleware API. This middleware constitutes the second component we have to provide. It has to integrate services such as group management, service discovery and routing protocols, event management, etc. For this purpose, we will take advantage of the work that we are carrying out in the context of the SMEPP project [11] (Secure Middleware for Embedded Peer-to-Peer Systems), a EU funded project (FP6 IST-5-033563), which has the general goal of developing a new

secure, generic and highly customizable middleware, based on a new network centric abstract model for EP2P systems.

As a further component we are interested in the development of a graphic support, which allows application programmers to construct graphs of their WSAN applications in a visual, interactive way. Therefore, graphs similar to the one shown in Figure 3 can be designed and then by means of an automatic code generation tool C#, Java or nesC files are generated that contain stubs for implementing the corresponding services of the graph.

6. Conclusions

A service-oriented framework for programming Wireless Sensor and Actor Networks has been proposed. It supports a high-level programming model that allows us to define the services and their interactions in order to build the applications in a platform independent way. The model considers group formation as a key issue in order to achieve scalability and efficiency in the systems. The abstract language syntax established has been detailed and an application example has been used in order to show its expressiveness and usability. We are currently involved in the development of three prototypes using different sensor technologies in order to evaluate the usability and performance of the approach on real systems.

References

[1] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal*, 2(4):351–367, 2004.

[2] A. Bakshi and V. K. Prasanna. Programming Paradigms for Networked Sensing: A Distributed Systems' Perspective. In *Proceedings of the IWDC'05*, pages 451–462, 2005.

[3] Crossbow. http://www.xbow.com/.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI 2003)*, pages 1–11, San Diego, California, USA, June 2003. ACM Press.

[5] J. Gehrke and L. Liu. Sensor-network applications. *IEEE Internet Computing*, 10(2), 2006.

[6] K. Henricksen and R. Robinson. Middleware for Sensor Networks: State-of-the-Art and Future Directions. In *Proceedings of the 1st International Workshop on Middleware for Wireless Sensor Networks (MidSens 2006), co-located with the 7th International Middleware Conference (Middleware'06)*, Melbourne, Australia, November 2006. ACM Press.

[7] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits. OASiS: A Programming Framewor for Service-Oriented Sensor Networks. In *Proceedings of the 2nd IEEE/Create-Net/ICST International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE'07)*, Bangalore, India, January 2007. IEEE Computer Society Press.

[8] N. Y. Othman, R. H. Glitho, and F. Khendek. The Design and Implementation of a Web Service Framework for Individual Nodes in Sinkless Wireless Sensor Networks. In *Proceedings of the IEEE International Conference on Computers and Communications (ISCC'07)*, pages 941–947, Aveiro, Portugal, July 2007. IEEE Computer Society Press.

[9] A. Rezgui and M. Eltoweissy. Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead. *Computer Communications*, 30:2627–2648, 2007.

[10] B. Rubio, M. Díaz, and J. Troya. Programming Approaches and Challenges for Wireless Sensor Networks. In *Proceedings of the 2nd IEEE International Conference on Systems and Networks Communications (ICSNC'07)*, Cap Esterel, Frech Riviera, France, August 2007. IEEE Computer Society Press.

[11] SMEPP. http://www.smepp.net.

[12] SunSPOT. http://www.sunspotworld.com.

[13] TinyOS. http://www.tinyos.net/.