A Service-oriented Middleware 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

Wireless Sensor and Actor Networks are potentially one of the most important technologies of this century. There are many fields where they can be used in order to develop varied and interesting applications: high security environments, environmental monitoring, objects and events detection, military surveillance and precision agriculture. On the other hand, the ease of programming is a major barrier to the adoption of this kind of system. Recently, different high-level programming abstractions and middleware have appeared as promising solutions. In this paper, a middleware is proposed for USEME, a Service-Oriented Framework focussed on the deployment of lightweight services in sensors and actors. The high-level model supported abstracts application programmers from implementation tasks such as discovery, communication, group formation or real-time constraints.

Keywords*: middleware, wireless sensor and actor network, service oriented architecture, high-level programming, framework*

1. Introduction

Wireless Sensor and Actor Networks (WSANs) constitute a new pervasive and ubiquitous technology and are currently one of the most interesting fields of research. Technological advances make it possible to design advanced sensors (colloquially referred to as "motes") that can be deployed in the environment in order to gather information about physical phenomena and report it back to actor devices. These can react appropriately, 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]. A recent report from market research firm ONWorld predicts that the global market for WSANs will grow tenfold by 2011. However, the same report also identifies the ease of programming as the major barrier to the adoption of WSAN technology.

The complexity of designing and implementing this kind of system makes the supply of higher-level abstractions of low-level functionality necessary as well as the middleware to support it 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 [6][10]. Some of them are based on the Service Oriented Architecture (SOA) [9][7][8]. In Service Oriented WSANs, node sensing and actuation capabilities are presented as in-network services. The application programmer can then specify the execution flow by simply compacting the appropriate services together.

In [2] we proposed USEME, a Service-oriented framework to develop WSAN applications. The high-level programming model and the corresponding abstract programming language were described, which abstract the application programmers from implementation tasks, such as discovery, communication and group formation. Priority, period and deadline issues are also taken into account in order to deal with real-time requirements at the service specification level. In this paper we introduce the middleware that has been designed for supporting USEME. The middleware is in charge of service publication and discovery issues, command and event invocation and communication, real-time constraints and group management. Two prototypes have been implemented in order to assess the feasibility of our approach, using Imote2 .Net technology from Crossbow [3] and Sun SPOT devices [11].

The rest of the paper is structured as follows. Section 2 gives an overview of the USEME Framework. The proposed Middleware is presented in section 3. Some implementation details of the prototypes developed are described in section 4 and finally, some conclusions are outlined in section 5.

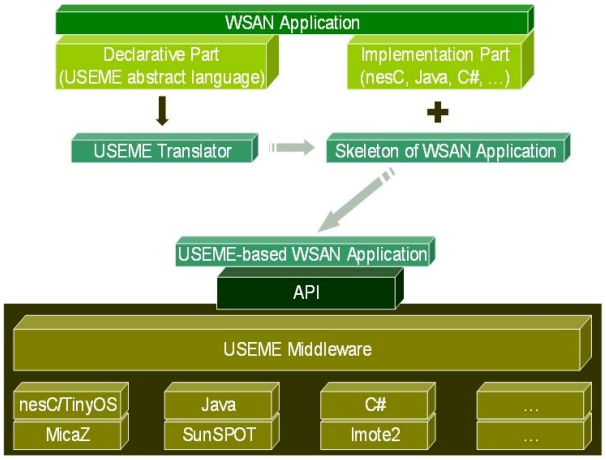

Figure 1. Framework Scheme

```
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;
}

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 2. Abstract language

2. USEME Framework

Figure 1 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 platform-independent declarative part where an abstract programming language is used to specify the main elements involved in the system: nodes (sensors and actors), groups and services in order to establish the global behavior of the application, using the service-oriented paradigm. By means of ports, which define asynchronous and synchronous commands and events, services interact with each other and can be composed to form complex ad-hoc systems. The declarative part is the input of a translator that generates the skeleton of the WSAN application. This part is common to every node in the network. In order to show the expressiveness of the abstract language, Figure 2 depicts an example where a sensor node publishes a temperature service in two different groups, the air conditioning control group and the water sprinkler control group, which can be integrated in a building monitoring and control application. A detailed description of the abstract programming language and the application is presented in [2].

- An implementation part where the programmers have to add the necessary code to the skeleton to implement the services defined in the declarative part. This implementation is platform-dependent, that is, the application programmer can use the component model provided by the nesC language [4] in order to be used on MizaZ-based motes [3] or object oriented languages such as Java or C# to be used on actor devices or new generation motes such as SunSPOT and Imote2.

The code obtained after developers fill the skeleton application will be the USEME-based WSAN application which will interact with the middleware through its API.

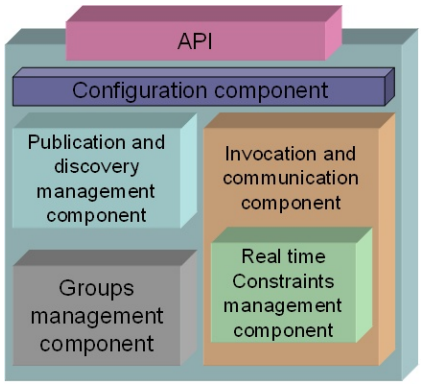

Figure 3. USEME Middleware

3. USEME Middleware

The proposed middleware provides facilities to address the different nodes in terms of logical, dynamic relationships among the services involved in the application. The use of a middleware makes applications running on the network safer and easier to deploy and update. Middleware ensures the proper use of the network by means of the components depicted in figure 3, which are described in the following sections.

3.1. Configuration

Configuration component allows application programmers to customize different parameters of the middleware.

These parameters configure the global behavior of the middleware and can lead to a better performance if they are specifically tuned by the application programmer. There are many options that can be changed using this configuration component such as deadline times, discovery frequency or maximum packet size.

3.2. Publication and discovery

The publication and discovery management component not only has to keep a record of the services published by the node but also of the services required (nodes that publish them). Therefore, it has to update the state of the network periodically in order to detect network topology changes.

In USEME middleware, publication is implemented using local timestamps. All nodes requiring a service make a broadcast request to determine which nodes provide this service. All nodes receiving this request will check whether they provide the service and if so, they will answer such petition. The node that initially made the request can then take note of the different available nodes. Besides storing the available node identifiers it stores a timestamp indicating the moment when that node was discovered. This timestamp allows services whose nodes are no longer available to be detected and erased from the available node list.

To achieve this goal a discovery thread is launched once a node is initialized, which carries out discovery and publication tasks from then on. To do so the thread wakes up periodically and sends discovery messages. If new messages of newly added nodes are received then available node list for every service they provide are updated. If a node receives a message from a node that was already in the available node list, its timestamp is updated to the current time. Finally if a node has not been updated for a certain period of time it will be erased from the list. This behavior is achieved using an additional thread which detects and erases old nodes which have not been updated. These two threads involved in discovery and publication are triggered periodically so node failures or additions are correctly handled.

There are two types of tables related to discovery and publication tasks (figure 4):

- **Provided services table**. This table appears in every node providing services and it stores all services provided by the local node.
- **Service location table**. Only nodes requiring services have this table. Node identifier, group and location of all nodes providing services to the local node are stored in this table. Application programmers can ask for a list of nodes providing a desired service. Middleware answers the petition by querying this table.

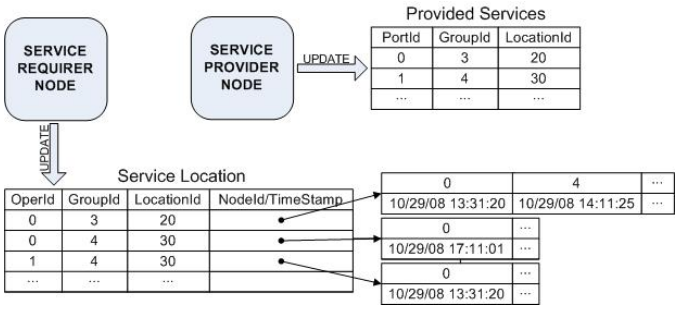

Figure 4. Example of service location and provided service tables

3.3. Invocation and communication

The use of a service-oriented architecture allows the application programmer to execute remote commands in the same way as if they were invoking local commands. The middleware in this case is responsible for the translation of these commands calls into communication packets between remote nodes. The invocation and communication component deals with this task.

In a USEME sensor network each node is uniquely identified with a node identifier. When an application programmer wants to execute a command in a remote node s/he has to specify the NODE_ID of such remote node. Packets sent or received in a node are dispatched through commands in the API. To simplify and structure the design of USEME middleware, an invocation and communication component is implemented with a single input and a single output point. That means all packets are sent using the same API commands and also that all are sent sequentially. The same also applies when receiving messages.

Either synchronous or asynchronous commands are used to invoke commands from remote services. Synchronous commands interrupt the program execution until command output parameters arrive from a remote node or a deadline expires. On the other hand asynchronous commands do not interrupt the program execution, therefore output parameters (if any) are received via events as long as time constraints are satisfied.

Once a node calls a command in another node the middleware translates this call in a command request packet which is sent to the destination. Upon arrival at destination the node reads the packets and executes the requested command. In the case of a synchronous command a result is generated and sent back to the source node. The source node must therefore wait for the answer to continue its normal execution. With asynchronous commands the source node does not have to wait for the answer and can continue doing other tasks. When the command result is ready, the

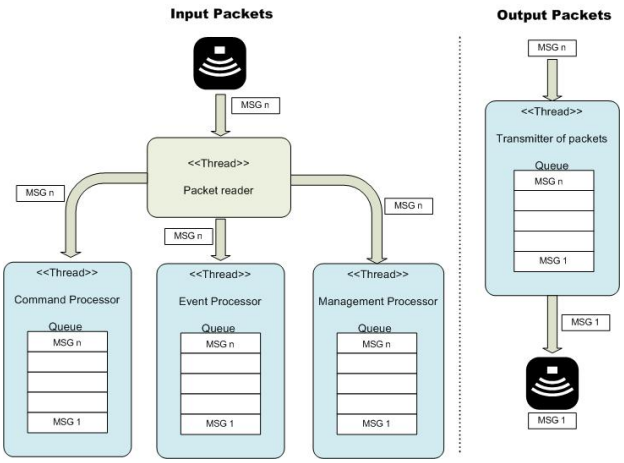

Figure 5. Packet sending and receiving

source node is interrupted via an event. The code attending to this event can then use the result to calculate whatever it wants.

Events are treated separately. That means that the execution of a command in a node does not prevent events from been signalled and processed. This is achieved using an additional events queue and thread. Nodes providing services are responsible for detecting when their events must be triggered. When an event is triggered the node sends a broadcast notification. All nodes requiring the service provided by the node triggering the event process that packet and execute the corresponding code attending the event. Events are often used to inform nodes that an abnormal situation has happened. Thus, an event must be able to be executed in a node even if it is already executing a command.

There are also other kinds of messages that need the same time constraints. In USEME middleware this restriction has been solved by using three separate queues: the events queue, the commands queue and an additional queue for the remaining packets such as publishing and discovery packets. Each queue is managed by a different thread. This architecture allows concurrency when processing those different kinds of messages. This whole architecture is depicted in figure 5.

An important issue when designing a communication system (especially when there are wireless devices) is the choice of an appropriate format for the packets that are going to be sent across the network. In our approach the format of the packets has been designed bearing packet traffic and efficiency in mind. One way to do this is by using command and type ids. We assign an id to every group, command or event in the network created. When sending packets we use that id instead of the actual name of the group, command or event. By doing so it is possible to achieve smaller packets and improve communication protocol efficiency.

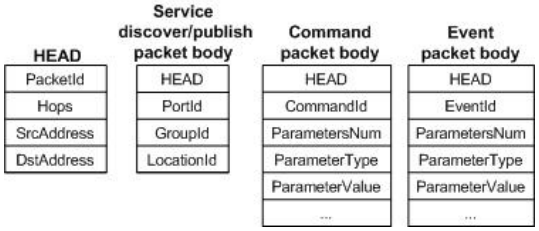

Figure 6. Communication Packets

A USEME packet is formed by a head and a body. The head format and the different body formats used are depicted in figure 6.

The fixed header is composed of 4 fields:

- **Packet id**: this predefined number identifies the packet type.
- **Hops**: indicates the maximum number of hops the packet can travel across the network. If the hop field of a packet reaches 0 the packet is destroyed if it has not yet reached its destination. Whenever a packet is received in a node the value of this field is decreased.
- **Source address**: specifies the address of the node which sends the packet.
- **Destination address**: specifies the address of the node which receives the packet.

Command, command answer, event and publishing and discovery packets have the following body:

- Command and command answer packets (either synchronous or asynchronous) have the same structure. They specify the different parameters used in the command request or command request response. This type of packet is composed of the header, the command id and finally the number of parameters and the actual parameters themselves. Before indicating each parameter we have to indicate its type. Again we use an id to specify each possible type.
- Event commands have the same structure as command packets.
- Discovering/publishing packet body is shorter than command and event packets. It only specifies the port, group and location of the service to discover or to record in the available node list. There is a packet for a discovery request and another one for the response.

CONSTRAINTS / OPERATION	Priority	Period	Deadline
Synchronous commands	Provided		Required
Asynchronous commands	Provided		Required
Events	Provided	Provided Required	

Figure 7. Possibilities the real time constraints

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

Figure 8. Real time constraints

3.4. Real time constraints

The middleware provides mechanisms to carry out the real time constraints which can be established through the USEME abstract language. These constraints can be applied to both events and commands. We can define the following constraints:

- **Priority**: Some activities are more important than others and should be executed first, enhancing this way the system response time.
- **Deadline**: It establishes the maximum execution time for a command.
- **Period**: It defines the time interval for a periodical event.

Figure 7 shows the different constraints that the application programmer can specify (using the abstract programming language) for each operation depending on the place they are declared, a provided or a required port. For example, a priority constraint can be attached to any operation but only for a provided port.

To control the priority, the middleware will assign a number, which is specified with the USEME abstract language, to each packet received (command or event). Therefore, packets are ordered in the corresponding queue depending on this number.

In the case of the deadline, when a node calls a command from another node, it first sends a message. Then the execution of the program waits for the answer the using sync mechanism. If the time specified by this mechanism expires before an answer is obtained, the command will raise an exception. Otherwise, the data received is handled.

Finally, the period is the easiest constraint to implement. As we see in figure 7 the period can only be established on the events which will be handled by different threads (a thread for each event) executing in a cyclical way each certain time (period).

```
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 9. Water Sprinkling group definition and creation

Thanks to the middleware the application programmer only has to worry about establishing the desired constraints on the provided or required commands and events for the services published by the nodes, using the abstract language. Figure 8 shows the definition of a temperature service, which provides a port with some real-time constraints.

3.5. Groups management

Group is a key concept for service composition and is one of the important issues in achieving scalability and efficiency. A group is formed by several nodes which share a common behavior. For example in a scenario where we are controlling and monitoring different parts of a building with a sensor and actor network a group could be the one dealing with the water sprinkler control. Figure 9 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 belongs to the group.

The middleware automatically manages the use of services inside the different groups. When forming a group all nodes which want to become part of it must communicate with the leader of the group and check whether they can join the group. The leader of the group can accept or reject the node depending on the group restrictions and node capabilities. For example if the cardinality of the group is set to

ten it means that no more than ten nodes can belong to the group.

4. Implementation

In order to verify the feasibility of our approach and evaluate its performance on a real system, we are currently working on two different prototypes.

The first implemented prototype uses the new generation Imote2 motes from Crossbow. They 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 which allows programmers to create code directly using C#.

The other implemented prototype has been developed using SunSPOTs. SunSPOT motes have 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.

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.

To fully implement the USEME framework we are also involved in the implementation of the USEME translator that takes as input the declarative part of an application (USEME abstract language) and generates a (Java or C#) program which uses the operations supplied by the middleware API.

On the other hand we are developing a graphic support, which allows application programmers to construct graphs of their WSAN applications in a visual, interactive way. By means of an automatic code generation tool the declarative part is generated.

5. Conclusions

In this paper, USEME middleware has been proposed. It allows programmer to simplify the use and programming tasks of wireless sensor and actor networks. On the one hand the programmer has to describe the services, their interactions and the groups of nodes involved in the application using a high-level abstract language. On the other hand, s/he has to implement the behavior of the services using the chosen programming language (C# and Java in our case) and the Middleware API. The different components supporting the middleware have been described. Two prototypes have been developed in order to evaluate the feasibility of this approach.

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] E. Cañete, J. Chen, M. Díaz, L. Llopis, and B. Rubio. Useme: A service-oriented framework for wireless sensor and actor networks. In *Proceedings of the Eighth International Workshop on Applications and Services in Wireless Networks (ASWN 2008)*, pages 47–53, Kassel, Germany, October 2008. IEEE Computer Society Press, CPS Conference Publishing Services.

[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] S. Hadim and N. Mohamed. Middleware challenges and approaches for wireless sensor networks. IEEE Distributed Systems Online, Volume 7 Issue 3, March 2006.

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