A Service-Oriented Approach to Facilitate Real-Time WSAN Application Development

Eduardo Cañete, Jaime Chen, Manuel Díaz, Luis Llopis, Bartolomé Rubio

University of Málaga, Dpto. Lenguajes y ciencias de la Computación. Calle de Bulevard Louis Pasteur, s/n. 29071 Málaga. Spain

Abstract
Due to the complex nature of developing Wireless Sensor and Actor Network (WSAN) applications it is obvious that new frameworks, tools, middleware and higher level abstractions are needed to make the task of the developers easier. Depending on the WSAN system we want to develop, different characteristics must be taken into account but, perhaps, some of the most important are the capacity to add real time constraints, the QoS and, of course energy saving. Our proposal USEME is a service-oriented and component-based framework which allows the easy combination of macro-programming and node-centric programming to develop real-time and efficient applications over WSANs. USEME allows the specification of real time constraints between services, permits the use of groups to structure the network and is platform independent.

Keywords: Middleware, Wireless sensor and actor network, Service Oriented Architecture, High-level programming, Real-Time, Framework

1. INTRODUCTION

A new world is emerging in which everything — people, mobile phones, cars, etc — are connected. All of this is called The Internet of Things. One of the most attractive and important parts of this kind of Internet is known as the Wireless Sensor and Actor Networks [1]. WSANs constitute a new pervasive and ubiquitous technology and are currently one 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”), which 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 altering the environment in order to tackle the problem. WSANs offer numerous advantages over traditional systems, for example, they make use of a large-scale flexible architecture (potentially hundreds or thousands of motes), high-resolution sensed data and application adaptive mechanisms. These unique characteristics make WSANs very useful for a wide range of application areas (medicine, security, critical infrastructures, intensive farming, …) and as a result the investment being made by industry, university and government organizations is considerable. A recent report from market research firm ONWorld predicts that the global market for WSANs will have grown tenfold by 2011. However, the same report also identifies the ease of programming as the major barrier for the adoption of WSAN technology. The effort needed to develop applications using this technology is enormous since we are talking about developing distributed applications (hundreds of sensors), taking into account the limited resources of the devices which form part of the network, i.e., the memory, computational power, energy, bandwidth, processing capabilities, limited transmission power, etc. Moreover, depending on how the devices interact with each
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. Therefore, 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 task of the application programmer.

In this work, we propose USEME (Ubiquitous SErvices for Mote Environments), a Service-oriented and component-based framework to develop real-time and efficient WSAN applications using a combination of macro and node-centric programming. In this way, the developers are not only able to specify the global behavior of the application (in a platform-independent way by means of the USEME abstract language) but they also have the possibility of developing the specific services provided by the different nodes using the language in which the devices are programmed.

The rest of the paper is structured as follows. In section 2 the motivation of our work is presented. Section 3 presents the USEME Framework and the different parts it is composed of are explained in detail. Finally, some conclusions are sketched and future work is discussed in section 4.

2. MOTIVATION

With the passing of time, the range of areas where the WSAN applications can be applied is constantly growing. We are in the presence of a relatively new, promising and powerful technology which offers a lot of advantages to develop many kinds of applications that were previously impossible. Not all are advantages however, as many parameters must be controlled. On the one hand, all WSAN applications have an important and common requirement: communication. Developers have to contend with a widely distributed application, therefore, coordination mechanisms for both sensor-actor and actor-actor interactions are needed. This communication requirement must be carefully considered by the developers before implementing WSAN applications, as both the network lifetime and scalability depend so much on it.

On the other hand, depending on the kind of application, other parameters must be controlled. For example in a building monitoring and control application, it would be important to add time constraints in order to deal with potentially dangerous situations in real time, such as possible fire alarms, gas leaks, intruder detection and so on. Figure 1(a) shows the different services this kind of system is normally required to perform:

1. Indoor environmental monitoring (heating, ventilation and air conditioning services).
2. Response to extreme events such as fire.

3. USEME FRAMEWORK

Figure 2 depicts the general scheme of the USEME Framework. Usually, the creation of a new WSAN application would follow the following steps:

1. A platform-independent declarative part where an abstract programming language (described in Section 3.1) is used to specify the main elements involved in the system: nodes (sensors and actors), groups, services and ports 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. This part is common to every node in the network.
2. The application global behavior established in the previous step passes through a translator which generates a platform dependent-skeleton of the application.
3. The skeleton must then be completed in order to program the behavior of the different services the application has. In this implementation part programmers have to add the necessary code to the skeleton to implement the services defined in the declarative part. It is carried out in the following way:

   (a) For every provided port in each service the programmer has to implement every synchronous and asynchronous command and also define the behavior of every event (when it has to be raised and the values it has to send).
   (b) For every required port in each service the programmer has to implement the code he/she wants to execute when an event is received (Event Listener).
   (c) For every service that has required ports the programmer has to implement the main program code which uses the operations (commands and events) of the required ports. This program is typically executed in actor devices and carries out the main task of the node.

This implementation part is platform-dependent, that is, the application programmer can use the component model provided by nesC [5] on MicaZ-based motes [6] or object oriented languages such as Java or C# on new generation motes such as SunSPOT and Imote2.NET.
4. Once the final application is obtained and compiled (for the specific platform) the resulting files generated are deployed in the different physical nodes of the network by uploading the corresponding code to each node.

5. Optionally, many different network parameters (such as deadlines, timeouts,...) can be configured in each running node to obtain optimal network performance.

3.1. The USEME abstract language

More and more, the tendency followed to develop software systems is to use high level languages, since they offer a much higher abstraction level. This positively affects the system and application development, by considerably reducing the development time. The development of WSAN applications is perhaps one of the areas where high level languages is most needed, due to the complexity of this kind of system. Nevertheless, the use of a high level programming language has its advantages and disadvantages. On the one hand, while the abstraction level is higher during the development process, it is obvious that the developers will have less control over the application they are developing. On the other hand however, time will be saved during the application development as the developers need not concern themselves with many difficult implementation tasks. The use of low level languages to program complex applications, is not only time consuming but also makes the code prone to errors and practically impossible to reuse.

In the framework proposed, a service composition high level language based on group hierarchy is offered. Figure 3(a) shows the UML Profile where our particular domain is modeled. By using this language the developer can specify the global behavior of the application using its basic elements: ports, services, nodes (actors or sensors) and groups. In addition, real time, group and reliability constraints can also be specified.

An application developed with this language is based on the following architecture: a particular location can have several groups composed of nodes with similar characteristics. Nodes publish their services within one or several groups and the services can interact with each other (within the same group) using the ports in order to form more complex services.

Figure 3(b) depicts an example of a graphical view where the basic elements appear. The WaterSprinklerGroup and AirConditioningGroup represent the “water sprinkler” and “air conditioning” groups deployed in a room, respectively. This way, the WaterSprinklerActor node is the actor device implementing the water sprinkler activity (service WaterSprinklerService), which requires both temperature and smoke services (services TempService and SmokeService, respectively). On the other hand, the AirConditioningActor node controls the air conditioning (AirCondition-
ingService), which requires both temperature and humidity services. The THSensor node is a sensor type supplying these two services (TempService and HumidityService) and it belongs to both groups.

The following sections offer a detailed explanation of the service composition language used to support our service-oriented model. Different code examples will be shown in order to clarify the explanation (only the water sprinkler group will be used as an example).

### 3.1.1. Ports

The ports are modeled by means of a port template. Within a port, three different operations can be defined:

1. **Asynchronous Command.** This operation allows us to execute remote commands in other nodes without interrupting the execution of the program in the node which makes the call.
2. **Synchronous Command.** This operation is similar to the above operation, the only difference is that it interrupts the execution of the program and waits for either the termination of the operation in the destination node or a timeout.
3. **Event.** By means of this operation, the nodes are able to raise events in order to inform other nodes.

Ports are the entity through which the different operations related with temperature and smoke data will be modeled. Thus, two different ports are specified as shown in the following code.

As shown in the figure 4, three operations are specified in the temperature port. Temp event is used to periodically send temperature levels to the air conditioning system. Another event called HighTemp is raised when the temperature rises up over a dangerous threshold. This event will inform the water sprinkler actor when a temperature threshold is exceeded.

The different event behaviors (periodical or not) are achieved adding real time constraints which will be explained later.

The asynchronous command SetThreshold is used to modify the threshold used by the HighTemp event. On the other hand, the smoke port only needs a synchronous command called GetSmoke which will be used by the water sprinkler to know the smoke level when it receives a HighTemp event.

### 3.1.2. Services

A service definition consists of a service description followed by the ports that form part of the service, allowing it to interact with other services. Service ports are classified into two categories:

- **Provided ports.** Ports offered by the service to other services.
- **Required ports.** Ports required by the service from other services.

The ports described in the previous section are provided or required through the SmokeService, TempService and WaterSprinklerService services. On the one hand, SmokeService and TempService provide SmokePort and TempPort ports respectively, that is, these services provide information. On the other hand, WaterSprinklerService requires the ports provided by the SmokeService and TempService and provides the FireAlarmPort.
Besides specifying if a port is provided or required, the constraints that must be satisfied by the operations of the services need to be specified in the service template. Figure 5(a) shows an example of how to define these constraints.

The `GetSmoke` operation defined in the `SmokePort` is specified so that it will be executed in a reliable manner.

In the `TempService`, it is only necessary to add constraints in two of its operations. On the one hand, the `Temp` operation is set with a period of 5 minutes (300000 ms). This means that the node that publishes this service in a group will inform the corresponding nodes of the temperature level every 5 minutes. On the other hand, `HighTemp` is specified as a reliable operation with a priority level of 1 and a deadline of 5 seconds. Thus, if after 5 seconds the node in charge of raising the `HighTemp` event does not receive a confirmation indicating that it has arrived at its destination, the event will be raised again. The priority in HighTemp indicates the importance of the operation. In the example, the operation is given the highest priority since the high temperature in the system must be dealt with as soon as possible.

In the `WaterSprinklerService` (figure 5(b)), both `TempPort` and `SmokePort` are required. Several constraints are also specified. As is explained in section 3.1.5, the system must carry out a matching process between the provided and required constraints in order to match services. The `Max` constraint specified in the `GetSmoke` operation does not affect the matching process. This constraint is used to receive the higher smoke level provided by the different nodes within the same group.

The `FireAlarmPort` is provided by the `WaterSprinklerService` to notify them that a fire has been detected and to instruct the nodes to activate the water sprinkler device remotely, where necessary.

### 3.1.3. Nodes: Actors and Sensors

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. The services that a node is going to publish and the group where they are going to be published are specified.

The code shown above (figure 6(a)) defines three node templates. The `THSensor` and `SmokeSensor` node templates both receive the parameter `loc` which indicates where the node will be located. Also, another attribute is specified, `Type` to indicate that they are able to sense humidity, temperature and smoke levels respectively. These attributes are used to control the group constraints. Finally, both nodes publish the `TempService` and `SmokeService` in the group `WaterSprinklerGroup` placed in the location received in the parameter `loc`. On the other hand, the `WaterSprinklerActor` template is defined to model a third node. This node publishes the `WaterSprinklerService` in the `WaterSprinklerGroup` and `FireAlarmGroup` groups placed in the locations `loc` and `loc1`, received as first and second parameters respectively. Thus, a group called `WaterSprinklerGroup` will exist where three different services are published, two of these provide information (`TempService` and `SmokeService`) and the third service (`WaterSprinklerService`) will require this
information.

3.1.4. Groups

As mentioned previously, the group is a key concept for service composition and is important for achieving scalability and efficiency. It adds a new level of abstraction in order 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. On the other hand, through the `SensorType` and `ActorType` construct, it is possible to define which nodes can belong to the group on the basis of the value assigned to the attribute `Type`.

- **Cardinality.** This allows us to specify the maximum number of sensors and actors belonging to a group.

The activity of the water sprinkler device will be managed by a group in which this device will be the leader. A water sprinkler will act on the basis of the information received from the temperature and smoke sensors belonging to the same group. In this case, because we want to model the control of a water sprinkler, a group called `WaterSprinklerGroup` has been created.

As shown in the above code (figure 6(b)), the `WaterSprinklerGroup` template has a parameter to indicate where the group will be located. It also has three attributes to control the group constraints. The first attribute is used to indicate that both sensors and actors can join the group. The second attribute establishes that only temperature and smoke sensor nodes can join the group. Finally, the cardinality attribute specifies that the group must have at most one actor and between zero and three nodes for both temperature and smoke sensors.

3.1.5. QoS

The abstract language allows the developer to establish real-time, group and reliability constraints for a command or event when a service is being defined. It provides the developers with the possibility of adding important functionalities to their applications (some of them have been shown in the previous pieces of code). The types of constraints which can be defined in each different group are the following:

**Real-time.** These constraints can be specified by means of the following keywords:
Figure 7: Priority message treatment

- **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, this allows us to establish the priority of both commands and events. For commands, the priority must be set in the required port since it is there where communication is initiated. In the same way, the event priority can only be used in provided ports. Messages in the queues will be ordered by priority so that more importance is given to the message in the queue with a higher priority. Priorities are global to the application and are specified with an integer number (the lower the value, the more priority it has). Figure 7 shows an example where a command and an event arrive simultaneously at node 4. Because HighTemp has higher priority it will be the first one that is forwarded from node 4.

- **Deadline.** It establishes the maximum execution time expected for a command and it is specified in the required ports. On the other hand, it can also be used in events defined in ports provided. It means that users receive an answer confirming that the event has arrived at its destiny, or not.

- **Period.** It defines the time interval for a periodical event. It can be established in both provided and required ports. The middleware must analyze the possible differences between a required and a provided period for an event in order to assess the possibility of matching.

**Groups.** Other interesting kinds of constraints 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.

- **Min, Max, Avg, Median, Std and Mode.** The command will provide the minimum, maximum, average, median, standard deviation or mode value respectively of all nodes which provide the command in the group.

- **List.** The command will provide the list of values of all the nodes which supply it in the group.

**Reliability.** QoS policy controls the level of reliability associated with data diffusion (events, command invocation, command parameters). By default, the middleware will act with a best-effort policy. However, the reliable keyword can be specified in order to be sure that data will arrive at its destination. The matching process for reliability uses a provided/required model where a service requiring a reliable operation cannot be matched to a service that only provides the operation using a best-effort policy.

The middleware will deal with the reliability taking into account the importance of the operations (events and commands) and the packet collisions. The first issue is controlled by using priority queues in each node. It allows us to schedule the packet delivery and reception in order to speed up the operations on the basis of their importance. To deal with the second issue, developers can define in the configuration template the number of times a packet can be transmitted, in case the first communication attempt is not successful. After a certain number of attempts (specified in the configuration template) the message will be sent using an alternative path chosen by the middleware. Although any of these measures ensure that the message is successfully delivered it effectively increases the probability of success.

In the case where reliability is used in combination with a deadline, the retransmission of the packet in case the first communication failed will be made as long as the deadline is satisfied. If the deadline expires and the message cannot be delivered, the middleware will notify the programmer as in the first case. If an acknowledgement
Table 1: QoS

<table>
<thead>
<tr>
<th>Priority</th>
<th>Period</th>
<th>Deadline</th>
<th>Reliable</th>
<th>Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous Command</td>
<td>Required</td>
<td>Required</td>
<td>Provided/Required</td>
<td>Required</td>
</tr>
<tr>
<td>Asynchronous Command</td>
<td>Required</td>
<td>Required</td>
<td>Provided/Required</td>
<td>Required</td>
</tr>
<tr>
<td>Event</td>
<td>Provided</td>
<td>Provided/Required</td>
<td>Provided</td>
<td>Provided/Required</td>
</tr>
</tbody>
</table>

Create Group WaterSprinklerGroup('"Lab301'');
Create Leader Actor
       WaterSprinklerActor('"Lab301'', ''Floor-1'');
Create Sensor TemperatureSensor('"Lab301'');
Create Sensor SmokeSensor('"Lab301'');

(a) Creation of Instances

Create Configuration
{
   JOIN_GROUP_PERIOD = 300000;
   MAINTENANCE_GROUP_PERIOD = 60000;
   LEADER_CHANGE_PROTOCOL_PERIOD = 300000;
   LEADER_SUBLEADER_COMMUNICATION_PERIOD = 600000;
   BATTERY_LEVEL_THRESHOLD_LCP = 20%;
   OPERATION_EXECUTION = Parallel;
   RETRANSMISSION_ATTEMPTS = 3;
}

(b) Configuration of the network

Figure 8: Instances and Configuration

packet arrives at the node which made the call after the timeout has expired, it must be ignored. This decision is made by means of timestamps. Of course, all these parameters (priority, attempt numbers and deadline) can be adjusted by the developers in order to achieve a better network performance. The more reliable the operation is, the more energy it will consume.

Table 1 summarizes QoS constraints, showing the different possibilities that can be specified for commands and events.

4. CONCLUSIONS

WSANs are a new and promising technology but in order to fully develop their possibilities, higher levels of abstraction are needed. SOAs (Service Oriented Architectures) have proven to be an efficient and inter-operable solution to communicate remote nodes. USEME is presented as a framework based on services which allows the programmers to create applications for WSANs without having to worry about low level and repetitive tasks present in all WSAN applications. The framework present interesting features that make it suitable for a wide range of applications. USEME is platform independent, a combination of macro-programming and node-centric programming and it has group hierarchy architecture with the possibility of defining group constraints. It also provides programmers with a high level abstract language for service definition and composition and real-time support that allows programmers to define QoS constraints in the communication between services.

5. REFERENCES