# A Service-Oriented Programming Model for Real Time WSANs

E. Cañete, M. Díaz, L. Llopis, B. Rubio
{ecc,mdr,luisll,tolo@lcc.uma.es}
Depart. Languages and Computing Science
University of Málaga, Spain

**Abstract – The increasing complexity of the wireless sensor and actor network (WSAN) applications together with the technological evolution of this kind of system has allowed higher level programming paradigms to be proposed for these networks. In this paper, a service based programming model for real time WSANs is proposed. It allows us to specify the services and their interaction in order to build applications. The model syntax is platform independent which will facilitate its use by automatic tools in the implementation phase. The real time issue is important in WSANs so the model allows us to specify real time requirements as service priorities, periods of service execution or deadlines.**

## I. INTRODUCTION

[*]The combination of recent technological advances in electronics, nanotechnology, wireless communications, computing, networking, and robotics has enabled the development of Wireless Sensor Networks (WSNs), a new form of distributed computing where sensors (tiny, low-cost and low-power nodes, colloquially referred to as "motes") deployed in the environment communicate wirelessly to gather and report information about physical phenomena [1]. WSNs have been successfully used for a variety of applications, such as environmental monitoring, object and event detection, military surveillance and precision agriculture [2][3].

One variation of WSNs that is attracting growing interest among researchers and practitioners is the so called *Wireless Sensor and Actor Networks* (WSANs) [4]. In this case, the devices deployed in the environment act not only as sensors able to sense environmental data, but also actors able to react and to affect the environment. For example, in the case of a fire, sensors relay the exact origin and intensity of the blaze to water sprinkler actors so that it can be extinguished before it becomes uncontrollable. Actors are resource rich nodes equipped with better processing capabilities, higher transmission power and longer battery life than sensors. Moreover, the number of sensor nodes deployed in a target area may be in the order of hundreds or thousands whereas such a dense deployment is usually not necessary for actor nodes due to their higher capabilities. In some applications, integrated sensor/actor nodes, especially robots, may replace actor nodes.

The progress taking place in the software technology for sensors including Java [5] or .NET platforms [6] allows us to propose high level programming models to develop applications. In this paper we propose a service-oriented programming model to facilitate the development of applications. It provides a higher level of abstraction, interoperability and reusability [7], that is, users can specify services without knowing which WSAN execute them and services of different provider could operate. The programming model is platform independent so automatic tools will be able to generate code for the different platforms. Additionally, the real time requirements of the system such as priorities, periods or deadlines can be specified. The model defines the *mote/actor* template (node) which will publish the set of services accessible in the network. The access points to the node services are the *ports* which define the commands and events that other nodes may require. Commands can be synchronous or asynchronous and events are always asynchronous. The *service* template will include both provided and required ports necessary to carry out the service execution. Additionally, the model defines the group concept to build node subsystems with common characteristics.

Recently, some work based on the service oriented paradigm has been carried out. In [7] challenges on service oriented sensor-actuator networks are presented. Oasis [8] is a programming framework that provides abstractions for service oriented sensor networks but it does not deal with real time issues. In [9] the service oriented architecture is applied to real world scenarios such as WSAN home security systems but it does not take into account timing requirements that may be useful for these systems. Atlas [10] is a service oriented sensor and actor platform that enables programmable pervasive services.

This paper is organized as follows: In section II the service oriented programming model is presented. In Section III a brief example is presented and Section IV presents some conclusions and future work.

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

## II. The Service Oriented Model

In this section the main characteristics are detailed. A syntax has been defined to specify applications meeting the model characteristics. It allows developers to specify applications in a platform independent way and then this specification to be mapped to the desired platform. Figure 1 shows the development methodology; the application requirements (functional and non-functional) together with the service model semantics build a service based application. It includes the application behavior and an underlying layer (scheduler) to meet the model semantics. For example, focusing on real time issues, the scheduler must ensure that the most critical event or command will be executed first taking into account the priorities assigned in the model. Additionally, it will implement the periodical actions defined. Finally, automatic tools will map the service based application specification to the target platform building a middleware between the application and the operating system (TinyOs, Java Virtual Machine or .NET).

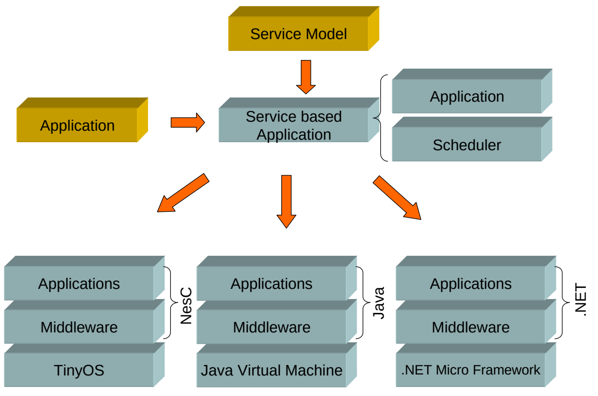

Fig. 1 – The development methodology

Fig. 2 shows a graphical example of a service-composition scheme. Optional ports are drawn in a dark color. Moreover, the ports and the service implementation are connected by means of dash-dotted lines. As we can see in the figure, the service provided by node A requires the ports offered by B and C and the port offered by D is optional. The service implementation will depend on the node characteristics and the supported technology.

### A. *Node Definition*

As commented previously, a node can be an actor or a sensor. In order to define a node we use a template (in the mote/actor being modeled) to specify which services want to be published and which group will be made.

```
Mote template MoteType1(i: Id, loc: Location){
  Id = i;
  Location = loc;
  Publish Service1, Service2 in group
    GroupA(loc);
  Publish Service1, Service2 in group
    GroupB(loc);
}
```

The code above shows the definition of a mote called *MoteType1* with two parameters: an identifier (*Id*) to be able to distinguish the motes/actors within a network and a location (*Location*) to know where the mote/actor is placed. Finally, two services (*Service1* and *Service2*) are published in two different groups (*GroupA* and *GroupB*).

To create a mote/actor based on this template, we will use the *Create* primitive:

```
Create mote MoteType1(1, "Room A");
```

This code will create a mote whose identity is 1 and it will be placed in room A.

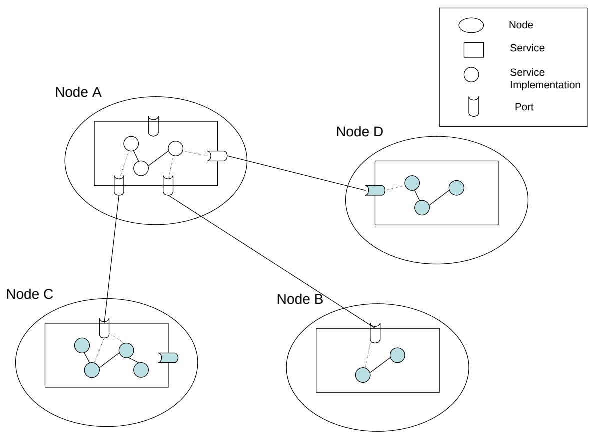

Fig. 2 – Service-composition scheme

### B. *Group Definition*

The group concept adds a new level of abstraction to join nodes with common restrictions. The model allows certain group constraints to be defined such as *Cardinality*, *DeviceType*, *MinBatteryLevelRequired*:

- **Cardinality**: We can specify how many sensors and actors must belong to a group.
- **DeviceType**: It defines what type of element (actors, sensors, or both) belongs to the group.
- **MinBatteryLevelRequired**: It indicates the minimum threshold battery that must have a sensor or actor to belong to a group.

```
Group MyGroup(loc :Location){
  Location = loc;
  Cardinality = (mote, 3-6) (actor, 1-1);
  DeviceType = "actor" or "mote";
  MinBatteryLevelRequired = 10.0;
}
```

Previous code shows the specification of a group where the parameter *Location* is mandatory. In addition, this group must have between three and six motes and one actor, and finally each sensor or actor trying to join the group must have a minimum battery level of ten.The model defines the *Create Group* template to create the group:

```
Create Group MyGroup(“Room A”);
```

This code will create a group of type *MyGroup* placed in the Room A.

### C. *Port Declaration*

Ports are the access points to node (mote or actor) services as well as being the mechanism used to compose the services offered to the system by the nodes. Within a port definition, developers can define commands (synchronous and asynchronous) and events which are asynchronous communication mechanisms. Ports are global definitions for all the services and therefore no implementation details are associated with them. This way, different services offering a specific port can be implemented in different ways: different execution platforms, programming languages, algorithms, etc. A simple port is defined below:

```
Port MyPort {
  CommandSync GetValue(out value :int);
  CommandAsync SetValue(in value: int);
  Event High;
}
```

*MyPort* defines a synchronous command named *GetValue.* It takes one output parameter, an asynchronous command named *SetValue*, which passes one input parameter, and an event called *High*.

### D. *Service Definition*

A service definition is composed of: firstly, a service description followed by the ports that will take part in the service. The port classification is divided into three types: *provided ports*, *required ports* and *optional ports*:

- **Provided ports**: Ports that the service offers to other services. The rest of the nodes can query this service through the provided ports.
- **Required ports**: Ports that a service needs in order to be executed. Other nodes in the network must offer these services.
- **Optional ports**: These ports are associated with non crucial services, e.g. ports used to monitor the system.

In the example below MyService provides M*yport* and requires *OtherNodePort*.

```
Service MyService {
  Description = “Example service”;
  Provides MyPort;
  Requires OtherNodePort;
}
```

### E. *Including Real Time Constraints*

Previous sections details how to build services based on applications but the model allows the developer the possibility of defining real-time restrictions for each command and event defined in the ports. These restrictions are specified by the keywords *Priority*, *Deadline* and *Period*.

- **Priority** allows us to define the priority of the commands and events defined in the service. It allows us to establish an order, attending to the most critical service first.
- The **Deadline** is established at the request of the service. It will allow the runtime system to analyze wether the requested service has been executed meeting the deadlines.
- The **Period** is defined in the events declared in the services but the model allows us to define a periodic request of the event. The period of the service establishes the minimum execution period. The run time system must analyze if the reception period of the event requested can be carried out.

The code below represents a service definition with timing constraints; firstly, we write a brief description, then, we define that the service provides a port (*MyPort*) and only one port is required (*OtherNodePort*). In this example, the service has no optional ports.

```
Service MyService {
  Description = “Example service”;
  Provides MyPort with constraints {
    GetValue Priority 5;
    SetValue;
    High Period 1000, Priority 10;
  }
  Requires OtherNodePort;
}
```

The *GetValue* command will be executed with a priority of 5 and the *High* event has an execution period of one-second and a priority of 10. The priority concept will allow us to solve the simultaneous execution of *GetValue* or the *High* event.

```
Service MyService2 {
  Description = “Example service2”;
  Requires MyPort with constraints {
    GetValue deadline 1000;
    SetValue;
    High Period 2000;
 }
}
```

From the point of view of the service request, a *GetValue* command is required but it must be executed before 1000 ms, *SetValue* does not have timing requirements and the *High* event is requested within a period of 2000 ms. As we can see in the service definition, it can be executed with a minimum period of 1000 ms so the run time system can be designed to adapt the event reception to the timing constraints required.

## III. An Example

In this section a brief example is presented to illustrate the use of different model templates. The example constitutes a real time WSAN to monitor and control fire in a building. We require four node types: the `FireDetect` node is an actor that is able to detect a possible fire and then act accordingly. The `TemperatureSensor` and `SmokeSensor` nodes provide temperature and smoke level measurements respectively. The monitor node (not specified in the example) receives data about the actions carried out by the fire detector and updates the temperature and smoke threshold values used to trigger the corresponding events. In the proposed system four ports exist, one for each service, although a service can usually provide more than one. The smoke port (`PSmoke`) offers a command to obtain the smoke reading and raises an event when this reading is higher than a pre-determined threshold. The temperature port (`Ptemperature`) is defined in the same way. The system is composed of only one fire detector, but there could be one or more temperature/smoke sensors per room and any number of monitors. In order to provide the fire detection service, at least one temperature sensor and a smoke sensor must be available. Therefore, temperature and smoke ports are required. Monitoring ports will be optional.

```
Mote template SmokeSensor (l : Location) {
  …
  Publish SSmoke;
}
Mote template TemperatureSensor (l : Location) {
  …
  Publish STemperature;
}
Actor template FireDetect (id: Location) {
  Publish SFireDetection;
}
Port PSmoke {
  CommandSync GetSmoke(out d: float ;out id:ID);
  Event SmokeHigh;
  Event SmokeData;
}
Port PTemperature {
  CommandSync GetTemperature(out d: float;
                             out id: ID );
  Event TempHigh;
}

Service Stemperature {
  Description=”Temp. Data”
  Provides PTemperature with constraints {
      GetTemperature priority 4;
      TempHigh priority 5;
  }
}
Service Ssmoke {
  Description=”Smoke Data”
  Provides PSmoke with constraints {
      GetSmoke priority 6;
      SmokeHigh priority 8;
      SmokeData priority 7, period 500;
  }

}
Service SFireDetection {
  Decription=”Detect fire”;
  Requires PTemperature with constraints {
    GetTemperature deadline 2000;
    TempHigh period 2000;
  }
  Requires PSmoke with constraints {
    SmokeData period 1000;
    SmokeHigh;
  }
}
```

Following the service specification, by using the priority, the execution order for simultaneous invocations to service commands or events can be determined when it is executed in the same node. In the example, event *SmokeHigh* is the highest priority action. Additionally, the fire detection requires the smoke data with a period of 1000 ms. Finally, the fire detection service will call command *GetTemperature* and its execution must carry out before 2000 ms.

The previous code specifies the service architecture but it is necessary to define the actors and sensors that will provide the services. It is carried out by means of the *Create* templates. From the point of view of actors the application will include the actor creation to provide the service. An actor *FireSDetect* is created with identifier 1 and location “building”.

```
Application {
  …
  Create actor FireDetect(1,”Building1”);
  …
}
```

The application from the point of view of sensors will create two services if sensor is able to get smoke and temperature data.

```
Application {
  …
  Create sensor SmokeSensor(1,”Building1”);
  Create sensor TemperatureSensor(1,”Building1”);
  …
}
```

## IV. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel service based programming model for real time WSANs. It allows us to define the services and their interactions in order to build the applications in a platform independent way. The real time constraints as priority, periodical actions or deadlines can be specified by means of the model. As future work a middleware to execute the model semantics will be carried out. A prototype is being developing to apply it to .NET or Java platforms.

## REFERENCES

[1] Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.,Wireless Sensor Networks: A Survey. Computer Networks Journal, 38, 4 (2002), pp. 393--422.

[2] Sensor Networks Applications. Special Issue of IEEE Computer, 37, 8 (2004), pp. 50--78.

[3] Wireless Sensor Networks. Special Issue of Communications of the ACM, 47, 6 (2004).

[4] Akyildiz, I.F., Kasimoglu, I.H., Wireless Sensor and Actor Networks: Research Challenges. Ad Hoc Networks J., 2, 4 (2004), pp. 351--367.

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

[6] http://www.xbow.com/Products/productdetails.aspx?sid=253

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

[8] M. Kuhwaha et al. OASiS: A Programming Framework for Service-Oriented Sensor Networks. Communication Systems Software and Middleware, 2007. pp 1-8

[9] J. Prinsloo et al. A Service Oriented Architecture for Wireless Sensor and Actor Network Applications. SAICSIT 2006. pp 145-154.

[10] J. King et al. Atlas: a service oriented sensor platform: hardware and middleware to enable programmable pervasive spaces. Proceedings of 31st IEEE Conference on Local Computer Networks, 2006, pp.630-638.