RTCoord: A methodology to Design WSAN Applications.

J.Barbarán, M.Díaz, I.Esteve, D.Garrido, L.Llopis, B.Rubio.
Depart of Languages and Computer Sciencie.
Málaga University.
{barbaran, mdr, esteve, dgarrido, luisll, tolo}@lcc.uma.es
29071 Málaga, Spain.
Corresponding Autor: Iñaki Esteve
(e-mail: esteve@lcc.uma.es; fax +34 952 13 13 97; phone: +34 952 13 28 65)

Abstract—Wireless Sensor and Actor Networks (WSANs) constitute an emerging and pervasive technology that is attracting increasing interest in the research community for a wide range of applications. WSANs have two important requirements: coordination interactions and real-time communication to perform correct and timely actions. This paper introduces a methodology to facilitate the task of the application programmer focusing on the coordination and real-time requirements of WSANs. The methodology proposed in this model uses a real-time component model, UM-RTCOM, which will help us to achieve the design and implementation of applications in WSAN by using the component oriented paradigm. This will help us to develop software components which offer some very interesting features, such as reusability and adaptability which are very suitable for WSANs as they are very dynamic environments with rapidly changing conditions. In addition, a high-level coordination model based on tuple channels (TC-WSAN) is integrated into the methodology by providing a component-based specification of this model in UM-RTCOM; this will allow us to satisfy both sensor-actor and actor-actor coordination requirements in WSANs. Finally, we present in this paper the design and implementation of an application which will help us to show how the methodology can be easily used in order to achieve the development of WSANs applications.

I. INTRODUCTION

Wireless Sensor and Actor Networks (WSAN) are one of the fields related to wireless sensor networks (WSN) which is gaining more and more interest from the research community. WSANs are a variation of WSN in which the devices deployed in the environment are not only the 'motes' capable of sensing different physical phenomena, but also actors capable of performing different actions depending of the data sensed by the sensors. In fact, they are capable of performing a reaction to the data sensed, in case it is needed. For example, we could use WSANs to develop an alarm system which could be installed in certain buildings, such as museums or banks. In case a burglar enters a room in the museum, he/she would increase the level of noise in the room.

The sensors deployed on it would relay the origin and intensity of the noise and send this data to the actors which could trigger and alarm, so that we could prevent the valuable paintings paintings in the museum to be stolen. Actors are usually equipped with higher computing and electronic resources than sensors. Furthermore, because of the actor's richer resources, the number of actors deployed in the environment will be much smaller than the number of sensors, which could be in the order of hundreds or even thousands. In some specific situations, integrated sensor/actor nodes, especially robots, may replace actor nodes.

Working with WSANs means we will have to deal with both sensors and actors. Thus, there is a need for coordination between sensors and actors and actors with each other. More specifically, sensor-actor coordination makes possible that sensors can send to actors the sensed data in the environment. Depending on the environment and the different situations we have to deal with, an actor may need every single reading from its area sensors or it might be interested only in receiving some critical information (for the example of the alarm system explained above, the actor might be interested in receiving only the measurements which have sensed a higher level of noise than usual). After receiving the data from the sensors, actors need to coordinate with each other in order to take decisions so that they can perform the most appropriate action according to the sensed data. Another important choice which must be taken is which actor will perform the action. In some situations, an action could be performed by different actors collaborating with each other in case it is very complex.

The real-time behavior issue will be a very important requirement in WSANs due to the fact that in many cases actors will need to react or receive information from sensors meeting timing requirements. As a result of it, both coordination models and communication protocols used in WSANs should support real-time properties.

The coordination and real-time issues will increase the functionality and capabilities of WSANs resulting in a greater software complexity for applications. Furthermore, sensor and actor programming is a tedious error-prone task usually carried out from scratch. These reasons prompted us to develop a methodology that facilitates the development of WSAN applications. We think that the component-oriented paradigm

is a good alternative[6] to design our methodology; because the software components[7] offer features such as reusability and adaptability which seem very suitable for WSANs as they are very dynamic environments with rapidly changing conditions. In order to show how the methodology can be easily used to achieve the design and implementation of WSANs applications we present an application which is in charge of the temperature and noise measurements for a set of buildings.

The methodology proposes to use a real-time component model (UM-RTCOM[8]) which will help us to specify the software components in our framework and will allow us to indicate some real-time constraints to them. In order to satisfy the above mentioned coordination requirements of WSANs, the methodology will incorporate a high-level coordination model called TC-WSAN[10].

UM-RTCOM is a previous development focused on software components for real-time distributed systems, and it is adapted to the unique characteristics of WSANs. TC-WSAN is based on TCMote [12][13] and it is integrated into the framework in order to satisfy the coordination requirements mentioned above. By means of special components, both sensor-actor and actor-actor coordination is carried out through tuple channels (TCs). A TC is a structure that allows one-to-many and many-to-one communication of data structures, represented by tuples. The priority issue, taken into account at two levels, channels and tuples, contributes to achieving the real-time requirements of WSANs. UM-RTCOM will allow us to give a component-based design for TC-WSAN so that we can take advantage of using the component oriented paradigm.

A. Operational Setting

Our reference operational setting is based on an architecture where there is a dense deployment of (not mobile) sensors forming clusters, each one governed by a (possibly mobile) actor. Communication between a cluster actor and the sensors is carried out in a single-hop way. Although single-hop communication is inefficient in WSNs due to the long distance between sensors and the base station, in WSANs this may not be the case, because actors are close to sensors.

Several actors may form a super-cluster which is governed by one of them, the so-called cluster leader actor. This way, a hierarchical structure may be achieved so that, in our operational setting, the base station can be considered as the leader actor of a cluster grouping the leader actors of outer clusters (Fig. 1).

Clustering avoids the typical situation in WSANs where multiple actors can receive information from sensors about the sensed phenomena. Lack of coordination between sensors may cause too many and unnecessary actors to be activated and as a result the total energy consumption of all sensors can rise. Moreover, clustering together with the single-hop communication scheme minimizes the event transmission time from sensors to actors, which contributes to support the real-time communication required in WSANs.

The rest of the paper is structured as follows. In Section 2 the main characteristics of UM-RTCOM are presented.

Section 3 introduces the methodology including the coordination model TC-WSANs with its main design goals, primitives and its component-based UM-RTCOM design. Section 4 describes the use of the proposed methodology in order to achieve the design and implementation of a real WSAN application. Finally, some conclusions are sketched in Section 5.
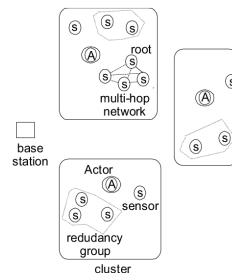

Fig. 1 Operational Setting

II. THE UM-RTCOM MODEL

Component-based development is a key technology in the development of modern software systems. In previous work, we presented UM-RTCOM, a component model especially suitable for real-time and embedded systems. Some of its main features have prompted us to use it for the development of WSAN applications. UM-RTCOM components are light-weight components which do not depend on any specific execution platform or heavy framework.

The model improves some features of standard component models, adding constructions to express temporal constraints, synchronization, quality of service, events, etc. It is a Hierarchical model where components act like containers of other components and, at the same time, provide interfaces. More details on the model characteristics are sketched in [8].

A. Component Types

There are two main component types: primitive and generic. Generic components are the standard components of the model. They provide services through interfaces and can require services of other generic components. On the other hand, primitive components (active or passive) are contained in generic components. They are the basis for building generic components, representing execution threads(active) or shared resources (passive).

B. Component Interactions.

Communication between components is performed through interfaces and events. UM-RTCOM provides synchronization primitives (wait, call, raise) which allow services and events to be invoked, raised or waited for.

C. Real-Time Characteristics.

UM-RTCOM was designed for use in real-time systems. It supports constructions for this type of system such as component configuration, specification of real-time constraints or the possibility of performing real-time analysis. We think WSAN systems can also benefit from some of these features.

III. DESIGNING WSANS APPLICATIONS WITH RTCOORD.

As it has already been explained in this paper, RTCOORD includes a coordination mechanism in order to achieve the sensor-actor and actor-actor coordination requirements. In our methodology we are using TC-WSAN, a coordination model based on tuple channels, in order to achieve these coordination requirements. In order to integrate TC-WSAN in our framework, we will give a component specification of this coordination model using UM-RTCOM. In figure 2, we show a component diagram for TC-WSAN.

The diagram includes the components provided by the framework used in the methodology and also the components actor and sensor which depend on the specific application we are developing and which must be supplied by the application developer.
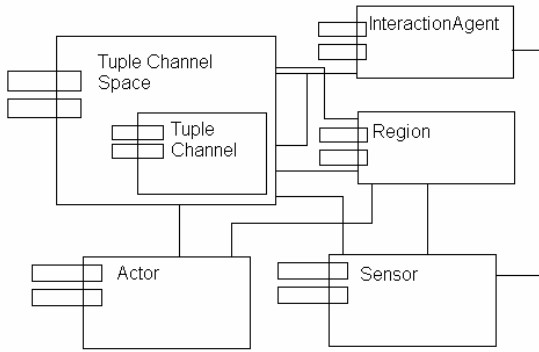

Fig. 2 Component Diagram: Framework

The region component is a passive UM-RTCOM that will help us to do the clustering while the Interaction Agent component is a passive UM-RTCOM component that uses the MoteIF interface supplied by Crossbow[18] which will allow us to communicate physically with the motes. The other two components in the diagram (Tuple Channel and Tuple Channel Space) will be explained in more detail later. Note that the `tcs` component contains the Tuple Channel component, so when actor and sensor components want to deal with a tuple channel, they will do it by means of the primitives provided by the tuple channel space component. In the next section, further details on the process of using the methodology in order to develop WSANs applications will be explained.

In the next subsections, more details on TC-WSAN and how it is used in our methodology are given.

A. The TC-WSAN Model.

A coordination model can be viewed as a triple (E,M,L), where E represents the entities being coordinated, M is the media used to coordinate the entities, and L is the semantics framework the model adheres to.

In our model, each cluster of the reference operational setting is considered as a Virtual Machine (VM) comprising these three items. The entities to be coordinated E are the nodes (actors and sensors). A Tuple Channel Space, which

constitutes the coordination media M, stores the tuple channels used to carry out the communication and synchronization between the sensors and the actor. Finally, the coordination "laws" L that govern the actions related to coordination are determined by the semantics of every model primitive. Further details on TC-WSAN are sketched on [10].

B. Using the Tuple Channel Space Component.

A Tuple Channel Space (TCS) is a shared data space accessed by the members of a cluster. The following UM-RTCOM component definition shows the API offered by the TC-WSANs model:

```
component CTupleChannelSpace {

struct attribute {
     string name;
     string value;
};
typedef sequence<attribute> channel_attributes;
typedef sequence<short> lchannel_id;
typedef sequence<any> tuple;

interface ITupleChannelSpace {

void create(in channel_attributes ca,
        out short chanid);
void get_attr(in short chanid,
     out channel_attributes ca);
void destroy(in channel_attributes
     search_pattern);
void find(in channel_attributes search_pattern,
        in float timeout, out short chanid);
// access to channnels
void connect(in short chanid,in string mode);
void disconnect(in short chanid);
void put(in short chanid,in tuple t);
void get(in short chanid,in short timeout,
        out tuple t);
}
input ITupleChannelSpace;
}
```

To facilitate the data-centric characteristics of sensor queries, attribute-based naming is the scheme selected [19]:

```
[attribute1 = value1, attribute2 = value2, ...]
```

In our model the channels are identified by means of an attribute-based data structure. This way, when a channel is created in the TCS its attributes are specified. For example, in case we want to create a many-to-one-channel so that the sensors can send the sensed data to the leader actor in the cluster, the attribute-based data structure identifying it could be the following:

```
[com_type = many_to_one]
```

One of the characteristics that contributes to achieving the real-time requirements demanded by WSANs is the priority issue. In our approach, a priority can be associated to a channel by means of an attribute. Then, a system entity can obtain the corresponding priority of a channel through the `get_attr` primitive.

On the other hand, when it is desirable that a channel be removed from the `tcs`, an attribute-based data structure with (probably) some partially specified fields (`search_pattern`) is used as an argument of the `destroy` primitive. A `search_pattern` is also used by the `find` primitive in order to find an appropriate channel to establish some communication. Primitives `connect`, `put`, `get` and `disconnect` will allow us to operate with the channels. These operations internally interconnect with component `CTupleChannel` described below.

C. Using the Tuple ChannelComponent.

A Tuple Channel is a priority queue structure that allows both one-to-many and many-to-one communication schemes. Both communication schemes are carried out in a single-hop way. Here, the priority issue affects tuples. A tuple is a sequence of fields with the form: $(t_1, t_2, ..., t_n)$ where each field t_i can be:

- a TC identifier.
- a value of any established data type of the host language where the model is integrated.

Entities access a TC by means of four primitives:

```
component CTupleChannel {
typedef sequence<any> tuple;
interface ITupleChannel {
  void connect(in string mode);
  // Producer (mode = P) Consumer (mode =C)
  void disconnect();
  void put(in tuple t);
  void get(in short timeout, out tuple t);
}
input ITupleChannel;
}
```

Before an entity can send/receive information through/from a TC, it must establish a connection by means of the `connect` primitive. When it no longer needs a channel, it executes the `disconnect` primitive in order to disable the TC connection.

A producer will use the `put` primitive to send information through a channel. Each time a `put` operation is executed, a new tuple is added to the channel. The way this tuple is treated and ordered inside the channel depending on the specified attributes. A consumer will use the `get` primitive to receive information (tuples) from a channel.

IV. APPLYING THE METHODOLOGY

In order to show how real WSAN applications can be developed by using the methodology proposed in this paper, we have developed a prototype where laptops play the role of actors and Crossbow family motes [18] are used as sensors. The laptops run Linux OS and the motes run TinyOS. The functionality which the application must provide is the sensing of the temperature and noise levels inside the three buildings constituting our Faculty in the University of Malaga and the generation of alarms in case the data sensed becomes critical.

Figure 4 shows the layout of the application, where we can see that three regions which have been created, one per each building; we will have only one actor in each cluster which will receive all the measurements from the sensors displayed in the building. The motes send the sensed data to the actors which must maintain a constant temperature for each building and trigger an alarm in case the level of noise sensed is higher than usual.

In figure 3, the design of the framework can be seen, as it is structured in layers. As it has been explained above, we are using TC-WSAN to satisfy the coordination requirements of WSANs applications and UM-RTCOM to specify the software components needed. In order to connect physically with the motes, the Interaction Agent component uses the MoteIf interface supplied by the TinyOS operating system.

In order to implement each component designed with UM-RTCOM, we will use Java. We will provide a mapping for each component designed with UM-RTCOM which will result in a single JAVA class for each component provided; this

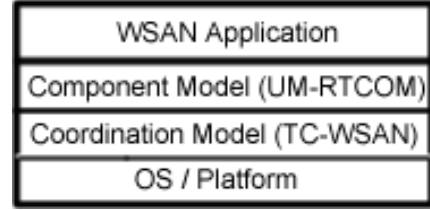mapping process could be done by an automatic mapping tool which we are currently developing.

Fig. 3 Framework: Layered Design

Depending on the component provided, it will be mapped to a Java thread (in case it is an active UM-RTCOM component) or to a Java class (in case it is a passive UM-RTCOM component). Figure 5 shows a component diagram which shows the component architecture we have used in the application. As it can be seen, there are some differences with figure 2, in which the architecture of the framework was shown. As it has been said before, the components provided by the methodology will be used by the actor and sensor components (specified by the user of the framework) in order to achieve the sensor-actor and actor-actor coordination requirements in WSANs.

The actor component will be responsible for creating the cluster by using the region component and will create a many-to-one tuple channel using the tuple channel space component primitives; this channel will allow the actor to receive the data sensed by the sensors. When it comes to defining the sensor component, it must be said that we have split the sensor component functionality in two, Noise Reader and Temperature Reader. This is done this way to clarify that noise reading will be more prioritary than the temperature readings.

Both components will use the `find` primitive provided by the tuple channel space in order to have access to the channel created by the actor component. Once they find it, they will connect to it as producers and then send the sensed data in the motes by using the `put` primitive of the tuple channel space component.

In figure 6, a sequence diagram is shown where all the components in the application and their interactions can be seen. These objects are the actor which will be the leader of the region (cluster) and so, will create it and ask the tuple channel space (`tcs`) to create a tuple channel which enables it to communicate with the temperature reader and the noise reader. Then the actor will connect to the tuple channel as a consumer, so that it can receive the data written in the channel by the `NoiseReader` and `TemperatureReader` components. Once the channel is created, both the `NoiseReader` and the `TemperatureReader` component will call the find primitive of the `tcs` in order to find the appropriate tuple channel which allows them to communicate with the actor component.

After finding the appropriate channel, both of them will connect to it as producers, so they can write the data sensed by the sensor network to the channel. Once they have connected to the channel, they will write the tuples in the channel with their priority, depending on each component (remember noise tuples will have a higher priority that the temperature ones), this will be done by using the `put` primitive. The `actor` will

read the tuples from the tuple channel using the `get` primitive; once it has read its value, it must check if the tuple read corresponds to a temperature or a noise tuple and then read its value and in case the value sensed is higher than usual, trigger an alarm.
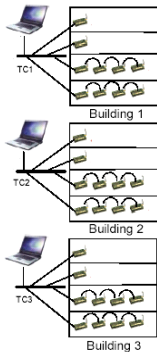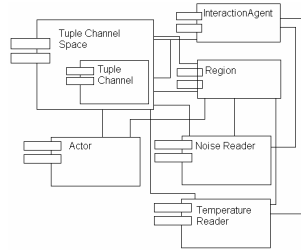

Fig. 4 Application Layout
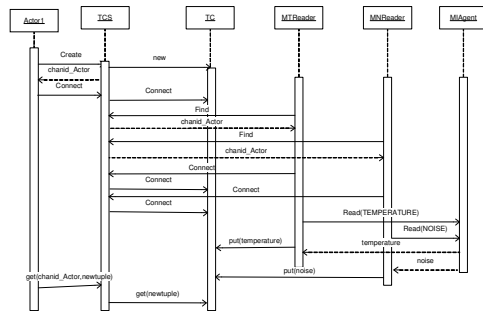

Fig. 5 Component Diagram: Application

Fig. 6 Sequence Diagram: Application

The following code shows the `Actor` component. The `Actor_Location` components periodically read from the channel associated to their respective buildings (by using `tcs`) to get data from the sensors. Note that the components which operations will be called are defined with the keyword `output`.

When we want to use a operation provided by an 'external' component, we use the keyword `call`. Note that the actor component will use the Tuple Channel Space as it was explained above in order to create a channel which attributes must be specified (in this case, we specify a many-to-one communication type channel and the location which could be any of the three buildings of the faculty, depending on the parameter location). Every time we read a tuple from the channel by using the `get` primitive provided by the `tcs`, we will use the Tuple component so that we know which type the tuple is (noise or temperature) and its value.

```
component Actor {
  output ITupleChannelSpace with tcs;
  output ITuple              with t;
}
component implementation Actor {
   Active Actor_Location {
     short chanid_Actor;
     Actor_Location (string Location) {
       channel_attributes ca; ca.length(2);
       ca[0].name="com_type";
       ca[0].value="many_to_one";
       ca[1].name="location";
       ca[1].value=Location;
       call tcs.create(ca,&chanid_Actor);
```

```
     call tcs.connect(chanid_Actor,"C");
   }
  void execute() {
    string type;
    int value;
    call tcs.get(chanid_Actor,INFINITE,&t);
    call t.GetType(t,&type);
    call t.GetValue(t,&value);
    if (type == "temperature")
    {
         if (value>MAX_TEMP)
    }
    if (type == "noise")
    {
         if (value>MAX_NOISE)
    }
  }
}
```

In order to achieve the implementation for the `actor` component, we show its mapping to a Java class. As the `actor` component is an active UM-RTCOM component, it must be mapped to a Java thread. All the components provided by the framework will be mapped to Java classes as they are passive UM-RTCOM components and every operation supplied by the UM-RTCOM components will be implemented as Java methods. So when we want to use an operation defined by an 'external' component, all we have to do is create instances of the classes we want to use and then call the appropriate method by using the classic '.' Java operator. The interaction among different objects in the application is shown in the sequence diagram shown above(Fig. 6).

```
class Actor implements Runnable
{
   TupleChannelSpace tcs;
   Tuple t;
     Int priority;
     Thread actorThread;
      Actor(string Location, int prioActor)
      {
    channel_attributes ca; ca.length(2);
        ca[0].name="com_type";
              ca[0].value="many_to_one";
       ca[1].name="location";
       ca[1].value=Location;
      priority = prioActor;
      tcs.create(ca,chanid_Actor);
      tcs.connect(chanid_Actor,"C");
      actorThread = new Thread(this);
      PriorityParams prioParams =
         New PriorityParams(priority);
  actorThread.SetSchedulingParameters(prioParams);
      }
     public void Run()
     {
      string type;
      int value;
      tcs.get(chanid_Actor,INFINITE,&t);
      t.GetType(t,&type);
      t.GetValue(t,&value);
      if (type == "temperature")
      {
        if (value>MAX_TEMP)
      }
      if (type == "noise"){
          {
            if (value>MAX_NOISE)
      }

      }
     public void start()
     {
          actorThread.Start();
     }
}
```

The components `MoteTemperature` and `MoteNoise` get the channel associated to their location (`find`) and periodically send the sensed data by using the `MoteTemperatureReader` and `MoteNoiseReader` active components. Note that the `priority` attribute allows us to establish the priority of the tuple (remember we considerered the noise tuples to be more

prioritary than the temperature ones). As the implementation of both is very similar, only the MoteNoise component will be shown.

```
component MoteNoise {
  output ITupleChannelSpace with tcs;
  output ITuple with tuple noise;
  output MoteInteractionAgent with miagent;
}
component implementation MoteNoise {
   short chanid_Actor;
   Int priority
   MoteTemperature(Int prio) {
   // Location and Connection to Actor
    channel_attributes fa; fa.length(1);
    priority = prio;
    fa[0].name="Location"
    fa[0].value=Get_Location();
    call tcs.find(fa,&chanid_Actor);
    call tcs.connect(chanid_Actor,"P");
   }
   Active MoteNoiseReader {
    void execute() {
      // Get data from environment
      call miagent.Read(SENSOR_NOISE, &noise);
      call noise.SetPrio(priority);
      call tcs.put(chanid_Actor,noise);
    }
   MoteTemperatureReader nReader with period 50;
}
```

As this is an active UM-RTCOM component, it will also mapped to a Java thread in order to implement it. In a very similar way as we did with the actor component, we obtain the following Java class:

```
class MoteNoise extends Thread
{
  TupleChannelSpace tcs;
  MoteInteractionAgent miagent;
    Short chanid_Actor;
    Int priority;
    Thread noiseThread
    MoteNoise(Int prio){
              channel_attributes fa;
            fa.length(1);
      fa[0].name="Location"
      fa[0].value=Get_Location();
      tcs.find(fa,&chanid_Actor);
              tcs.connect(chanid_Actor,"P");
              priority = prio;
              noiseThread = new Thread(this);
              PriorityParameters prioParams =
              New PriorityParameters(priority);
        noiseThread.SetSchedulingParameters(prioParams);
    }
    public void Run()
    {
        tuple noise;
        // Get data from environment
        miagent.Read(SENSOR_NOISE, &noise);
        tcs.put(chanid_Actor,noise);
    }
    public void start()
    {
        noiseThread.Start();
    }
}
```

V. CONCLUSIONS

In this paper we have presented a methodology to develop WSANs applications focusing on the coordination mechanisms and the real time requirements. The methodology proposed meets the component-based paradigm and uses UM-RTCOM, a component model which allows us to specify the software components constituting our framework.

We have applied the proposed methodology to a real example of a WSAN application in which the levels of noise and temperatures are sensed in the three different buildings constituting our faculty in the University of Malaga. In case the data sensed in higher than usual, the actors will trigger alarms. We have used laptops as actors and Crossbow motes as sensors. We have deployed several actors on each building and a single actor on each building.

As future work, we are currently developing automatic tools to map UM-RTCOM specifications to Java and we are also interested in using the Real Time Specification for Java

REFERENCES

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, "Wireless Sensor Networks: A Survey", *Computer Networks Journal*, vol. 38, no. 4, pp. 393–422, 2002.

[2] Sensor Networks Applications, *Special Issue of IEEE Computer*, vol. 37, no. 8, pp. 50–78, 2004.

[3] Wireless Sensor Networks, *Special Issue of Communications of the ACM*, vol. 47, no. 6, 2004.

[4] I.F. Akyildiz, I.H. Kasimoglu, "Wireless Sensor and Actor Networks: Research Challenges", *Ad Hoc Networks J.*, vol. 2, no. 4, pp. 351–367, 2004.

[5] D. Estrin, R. Govindan, J. Heidemann, S. Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks", in *Proc. MobiCom 1999*, 1999, pp. 263–270.

[6] J. Blumenthal, M. Handy, F. Golatowski, M. Haase, D. Timmermann, "Wireless Sensor Networks - New Challenges in Soft. Engineering", in *Proc. ETFA 2003*.

[7] G.T. Heineman, W.T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, 2001.

[8] M. Díaz, D. Garrido, L. Llopis, B. Rubio, J. M. Troya, A Component framework for Wireless Sensor and Actor Networks. 11th IEEE International Conference on Emerging Technologies and Factory Automation. IEEE Computer Society Press, pp. 300-307. Prague (Czech Replubic), September 2006.

[9] J. Barbarán, M. Díaz, I. Esteve, D. Garrido, L. Llopis, B. Rubio, A Real-Time Component-Oriented Middleware for Wireless Sensor and Actor Networks. International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2007).to appear. IEEE Computer Society Press. Vienna (Austria), April 2007.

[10] J.Barbarán, M.Díaz, I. Esteve, D. Garrido, L. Llopis, J.M. Troya and B. Rubio, *TC-WSANs: A Tuple Channel based Coordination Model for Wireless Sensor and Actor Networks* IEEE Symposium on Computers and Communications (ISCC'07) to Appear. IEEE Computer Society Press. Aveiro (Portugal), July 2007.

[11] M. Díaz M., D. Garrido, L. Llopis, F. Rus, J.M. Troya, "Integrating Real-Time Analysis in a Component Model for Embedded Systems" in *Proc of the 30th IEEE Euromicro Conference*. 2004, pp. 14-21.

[12] M. Díaz, B. Rubio, J.M. Troya, "TCMote: A Tuple Channel Coordination Model for Wireless Sensor Networks", in *Proc. ICPS 2005*, 2005, pp. 437–440.

[13] M. Díaz, B. Rubio, J.M. Troya, "A Coordination Middleware for Wireless Sensor Networks" in *Proc SENET 2005*, 2005, pp. 377–382.

[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems" in *Proc PLDI 2003*.

[15] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, G.P. Picco, "TinyLime: Bridging Mobile and Sensor Networks through Middleware" in *Proc. PerCom 2005*, 2005, pp. 61–72.

[16] T. Melodia, D. Pompili, V.C. Gungor, I.F. Akyildiz, "A Distributed Coordination Framework for Wireless Sensor and Actor Networks" in *Proc. Mobihoc 2005*, 2005.

[17] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms", *Mobile Networks and Apllications J.*, vol. 10, no. 4, pp. 563-579, 2005.

[18] Crossbow TechnologyInc: http://www.xbow.com

[19] The Real Time Java Specification(RTSJ) http://www.rtsj.org