# A Real-Time Component-Oriented Middleware for Wireless Sensor and Actor Networks

J. Barbarán, M. Díaz, I. Esteve, D. Garrido, L. Llopis, B. Rubio
{cjavibs,mdr,esteve,dgarrido,luisll,tolo}@lcc.uma.es
Depart. Languages and Computing Science
University of Málaga, Spain

***Abstract*— Wireless Sensor and Actor Networks (WSANs) constitute an emerging and pervasive technology that is attracting increased interest for a wide range of applications. The increasing functionality of this kind of network makes it necessary to propose tools and methodologies to facilitate software development. This paper proposes a middleware called MWSAN to provide a set of high level services for sensor and actor networks. The middleware meets the component-oriented paradigm and developers can configure it depending on the actor and sensor resources. It takes into account issues such as the network configuration, the quality of service (QoS) and the coordination among actors. MWSAN is specified in UM-RTCOM, a component model oriented to real time systems adapted to WSANs that allows us to take into account real time requirements in the applications. Additionally, we present an implementation model based on RT Java for actors. In order to apply the proposed middleware we describe an example specified in UM-RTCOM and its RT Java implementation.**

## 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 are not only sensors able to sense environmental data, but also actors able to react by affecting the environment. For example, in the case of a fire, sensors relay the exact origin and intensity of the fire 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 increasing expectations and demands for greater functionality and capabilities from these devices often result in greater software complexity for applications. Sensor and actor programming is a tedious error-prone task usually carried out from scratch. In order to facilitate the software development of this kind of system new tools and methodologies must be proposed. Middleware can simplify the creation and configuration of WSAN applications offering a set of high level services.

The component-oriented paradigm seems to be a good alternative [5] to define middleware. Software components [6] offer several features (reusability, adaptability, ...) very suitable for WSANs which are dynamic environments with rapidly changing situations. Moreover, there is an increasing need to abstract and encapsulate the different middleware and protocols used to perform the interactions between nodes.

However, the classical designs of component models and architectures (.NET, COM, JavaBeans) either suffer from extensive resource demands (memory, communication bandwidth, ...) or dependencies on the operating system, protocol or middleware. For this reason we propose to use UM-RTCOM [7], a previous development focused on software components for real-time distributed systems adapted to the unique characteristics of WSANs. The model is platform independent and uses light-weight components which make the approach very appropiate for this kind of system. In addition UM-RTCOM uses an abstract model of the components which allows different analysis to be performed, including real-time analysis.

In order to facilitate the development of WSAN applications we propose a novel middleware called MWSAN. It is specified with UM-RTCOM allowing us to define real time characteristics such as, establishing timing requirements (priority, periods,...) in the services and then to improve the temporal behavior of applications, attending to the most critical events first. MWSAN is composed of several components that provide a set of primitives related to sensor/actor interconnection, quality of service (QoS) and the actor coordination. These issues are very important and necessary and we think that high level primitives will simplify the work of the designers. The middleware is highly configurable in order to fit in with the limitations of sensors and actors. For example, the middleware for motes does not include the component for actor

This work is supported by EU funded project FP6 IST-5-033563 and the Spanish project TIN2005-09405-C02-01

coordination thereby reducing the required memory space. In order to implement our proposals, automatic tools will map the UM-RTCOM specification of MWSAN to RT Java [19] source code using an implementation called jRate [8] for actors. jRate allows us to meet the timing specifications of UM-RTCOM, implementing a priority based application where the highest priority events received by actors will be executed first. In the case of motes, due to the limited resources, tools will map MWSAN to nesC [9], a component language for sensors on TinyOS[10].

This paper is organized as follows: To finish this section, some related work is discussed and our reference operational setting is described. In section II we present the main characteristics of UM-RTCOM. In section III we propose MWSAN middleware using the UM-RTCOM component model. Section IV deals with implementation issues. Section V applies our proposals to a real example and finally, some conclusions and future work are presented.

### A. Related Work

Some approaches have incorporated the component-oriented paradigm to deal with WSAN programming. The most used is possibly nesC. This is an event-driven component-oriented programming language for networked embedded systems. Our proposal, UM-RTCOM, presents a higher level concurrency model and a higher level shared resource access model. In addition, real-time requirements of WSAN applications can be directly supported by means of the model primitives and the abstract model provided, reflecting the component behavior, which allows real-time analysis to be performed. Other languages such as Esterel [11], Signal [12], and Lustre [13] target embedded, hard real-time, control systems with strong time guarantees but they are not general purpose programming languages.

Much work has targeted the development of coordination models and the supporting middleware in the effort to meet the challenges of WSNs [14] [15] [16] [17]. However, since the above listed requirements impose stricter constraints, they may not be suitable for application to WSANs. In [18] UM-RTCOM is used to specify a coordination model based on tuple channels. The use of this component model allowed us to include real time characteristics in the tuple channels improving the application performance.

### B. Operational Setting

Our reference operational setting is based on an architecture where there is a dense deployment of stationary 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. Our approach, however, does not preclude the possibility of having one of these sensors acting as the “root” of a sensor “sub-network” whose members communicate in a multi-hop way. Therefore, this root sensor cannot send only its sensor measurements to the actor but also the information collected from the multi-hop sub-network. On the other hand, several sensors may form a redundancy group inside a cluster. The zone of the cluster monitored by a redundancy group will still be covered in spite of failures or sleeping periods of the group members.

Several actors may form a (super-)cluster which is governed by one of them, the so-called cluster leader actor. It takes centralized decisions related to the task assignment or QoS for the actors. Moreover, clustering together with the single-hop communication scheme minimizes the event transmission time from sensors to actors, which helps to support the real-time communication required in WSANs.

## 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. Instead, UM-RTCOM components are developed in a platform independent way and are later deployed in specific platforms as executables or libraries with a minimum overhead.

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 as containers of other components and, at the same time, provide interfaces.

### A. Component Types

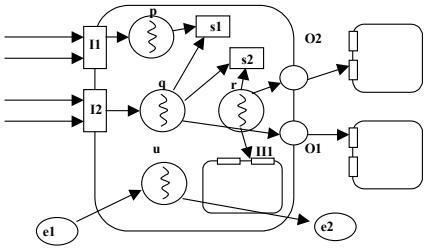
There are two main component types: primitive and generic. Generic components are the standard components of the model. They provide services through interfaces and may require the services of other generic components. On the other hand, primitive components (active or passive) are contained in and are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

*1) Generic Components:* These components are the basis of the model. They act as containers of other components, generic or primitive, and they can be composed of other generic components in order to complete their functionality.

A generic component has a public definition part with the provided and required interfaces, and an implementation part which includes the implementation of the services offered. The model distinguishes between input interfaces (provided) and output interfaces (required). The interfaces are defined using a CORBA-based Interface Definition Language (IDL).

UM-RTCOM also allows events to be used through the declaration of produced and consumed events. In this programming-style, a component declares which events produce and which consume. This way, components can communicate with each other without common interfaces.

Fig. 1 shows an example of a generic component with the declaration of input and output interfaces, events consumed and produced and the primitive components included.




```
Component comp_name {
  interface I1 {
      void op1(<parameters>);
      void op2();
      }
  ...
  input     I1,I2;
  output O1 with o1;
  output O2 with o2;
  consumes event_type1;
  publishes event_type2;
}
Component implementation comp_name {
  MyActive p,q,r,u;
  MyPassive s1,s2;
...
}
event event_type1(<parameters>);
event event_type2(<parameters>);
```

Fig. 1. Generic Component

*2) Active Components:* These components are primitive elements used to to express ”execution flows” inside a generic component. Concurrency is an important factor in real-time systems, so we use first-order elements to model it. In addition, the use of these components is also motivated by the later analysis phases.

Active components are responsible for the execution flow inside generic components through the interaction with other elements such as passive or generic components. Active components are also responsible for the treatment of the invocation requests on the generic container. Thus, the response to incoming requests is delegated to these primitive components.

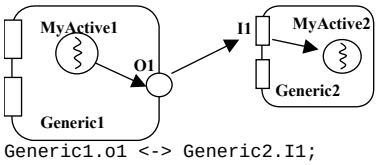
The use of an active component requires a definition part and a declaration part. In the definition part, the component defines a special ”execute” method to be invoked by the system in different ways: time-triggered, event-triggered, service requests, etc.

*3) Passive Components:* Shared resources are another important element in embedded and real-time systems. Passive components are primitive elements which allow us to use shared resources in the model. Basically, they cannot initiate any action and offer some basic services which can be invoked from active components. This behavior is used in order to facilitate later analysis phases. Passive components provide mutual exclusion with priority ceiling mechanisms which limit priority inversions.

### 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.

*1) Wait Primitive:* This primitive is used to wait for new invocations on services or for the creation of consumed events. It is used in active components and its syntax is the following:




```
Generic1.o1 <-> Generic2.I1;
```

```
interface I1 {
      void method1(in short a1,
       in short a2,in short a3);
      void method2(in short a1);
}
```

```
Active MyActive1 {
  void execute() {
    short a1,a2,a3;
    a1=1; a2=2; a3=3;
    call o1.method1(a1,a2,a3);
    call o1.method2(a3);
    }
}
```

```
Active MyActive2 {
  void execute() {
    short var_aux;
    wait I1.method1(a1,a2,a3)
            {var_aux=a1+a2+a3; }
         I1.method2(arg1);
            { var_aux=arg1+1; }
    }
}
```

Fig. 2. Synchronization Primitives

```
wait [<interf_name>.<method>|<event>](<args>)| Time
                <time> <blk>
```

The active component which invokes this primitive will be suspended until a call on the desired method occurs or an event is consumed. In addition, a timeout can be specified to avoid infinite waiting.

*2) Call Primitive:* It is used to invoke services of generic components. It can be used in active or passive components and its syntax is the following:

```
call <ref>[.<interf_name>].<method>(<parameters>)
                [Time <time>]
```

The developer has to use ”references” of other generic components, using the services provided in the input interfaces. The `call` primitive can be synchronous or asynchronous depending on the method definition in the interface.

*3) Raise Primitive:* This primitive is used to raise events asynchronously. When a component raises an event, this event is caught by all the components consuming that event. Its syntax is the following:

```
raise <event>(<parameters>)
```

Fig. 2 shows an example where two generic components (`Generic1` and `Generic2`) are interconnected. The example shows how the active component `Generic1::MyActive1` calls the method `I1::method1` provided by `Generic2` where the request is attended to by `Generic2::MyActive2` through the `wait` primitive.

### 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.

*1) Configuration Slots:* An important issue in component-oriented programming is reusability. UM-RTCOM allows the adaptation of components to different environments through the use of configuration slots. A configuration slot is a section in the component definition that is public to the user (as public interfaces are). The component user must supply values for all the parameters in the configuration slots. This way, we can adapt the component to new environments.

*2) Real-time Constraints Specification:* The user can indicate real-time constraints in the components. This is a very important element which is not included in other component models. This way, the user specifies the requirements regarding how a component is used.

The only visible elements of a component are the interfaces and events. The user can specify temporal constraints for methods or events indicating the minimum period between two invocations or a deadline for completing the request. The syntax has the following format:

```
instance_name constraints <Interface.Method>
     Period T, Deadline T ';'
constraints <event_type> Deadline D, Period T ';'
```

## III. Designing WSANs with UM-RTCOM

In this section we detail the main characteristics of MWSAN. The middleware provides a set of components to establish the network configuration, event synchronization and QoS, but new components can be added depending on the existing resources. Sensors and actors will be interconnected by means of the middleware but the design for sensors and actors is different. Figure 3 shows the different layers that compose the proposed middleware for actors. The highest level layer is composed of three types of components:

- Components for the interaction with sensors,
- components for the coordination among actors and
- components for the establishment of QoS among sensors and actors.

The communication and interaction between sensor/actors and actors/actors is carried out by means of the interfaces of these components. Additionally, the number of sensor interaction components will depend on the event types that an actor can deal with, i.e. if an actor can receive data related to temperature, light and noise then three components (one for each information type) will be included. The specification of our middleware in UM-RTCOM allows us to define a priority based system where the highest priority method is executed.

The next layer is composed of a set of priority queues to classify the methods and to guarantee the ordered delivery of the actions. The different components are mapped to a set of real time threads. Our middleware makes use of the OS scheduler to decide which action to execute taking into account the queue status. In order to carry out the communication from actor to a sensor, MWSAN implementation detects the sensor connections and assigns a sensor identification. A special primitive called `connected` can be used to detect the sensors or actors connected to the actor. This way actors can invoke operations in the sensors or in other actors.

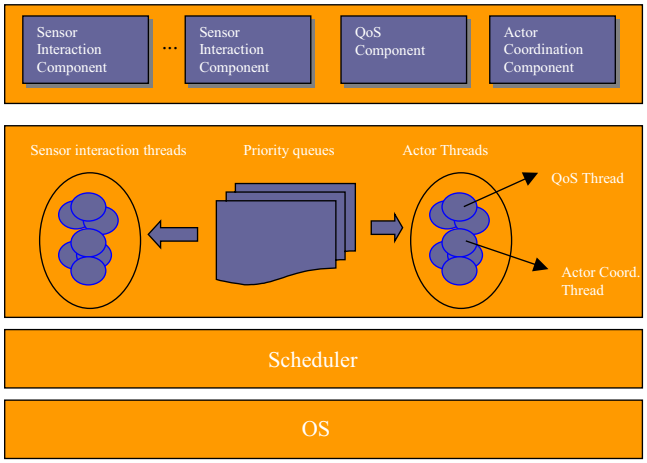



Fig. 3. The Middleware for actors

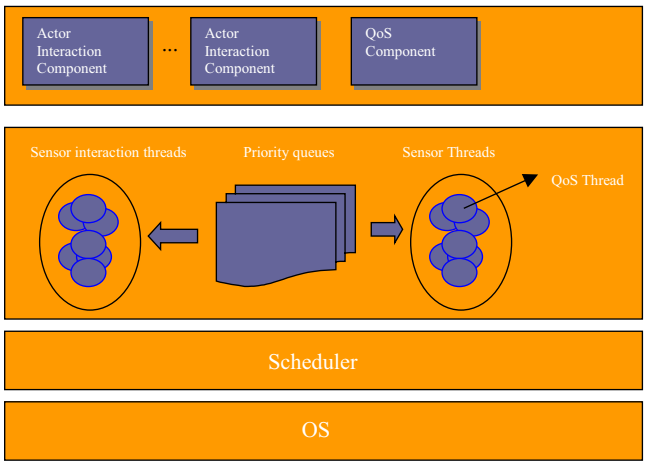



Fig. 4. The Middleware for sensors

Figure 4 shows the middleware for sensors. The design is similar to that for the actors but it does not include the components and threads necessary for actor coordination. It is composed of:

- Components for the interaction with actors and
- components for the establishment of QoS among sensors and actors.

Figure 5 shows the basic behavior for an actor. We define a thread to control the data input/output. A different "port" is configured for different data types received from sensors, the QoS information interchange among sensors/actors and actors/actors and the coordination among actors. Every action (method) to execute is stored in the priority queues that will be used by the scheduler to schedule the execution order of the threads in the actor.

In the following sections we detail the different parts of the middleware describing the primitives that will help designers in the application development.

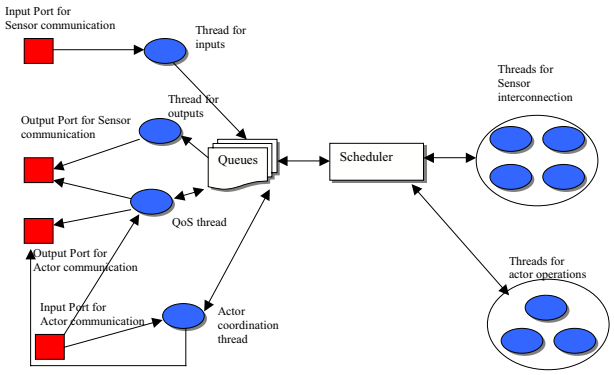



Fig. 5. Middleware basic behavior

### A. Sensor Interaction Components

This type of component belongs to the middleware defined in the actors. The developer will build a component for each event type that an actor can receive. For example, if an actor receives temperature and light data, then the developer will specify two components to interact with this data from sensors.

The problem of event synchronization is an important issue addressed in [4]. An actor can receive information of different event types from different sensors or information of the same event type from different sensors and it is necessary coordinate the reception to ensure ordered delivery. The use of the UM-RTCOM interfaces allows us to propose a classification method where each event type is encapsulated in a different component. From a low level implementation point of view, the same event type will be stored in the same queue associated with the component in order to guarantee the execution in the same order as it was received. As will be described in section III-C, this order can be changed making use of the priority definition in the methods. This is important because we can improve the temporal behavior of the system attending to the most critical events first.

Let us suppose an actor can receive information related to temperature and light from sensors. The developer will specify two UM-RTCOM components in the actor providing an interface with the primitives associated to each event type:

```
Component Temperature {
 interface IEv {
    void TempData(in float dat);
    void TempLow(in float dat);
    void TempHigh(in float dat);
 }
}

Component Light {
 interface IEv {
    void LightData(in float dat);
    void LightLow(in float dat);
    void LightHigh(in float dat);
 }
}
```

The implementation for these components would include the reception from sensors of normal data or critical data such as low or high temperature or light.

```
Component implementation Temperature {
  Active myTemp {
    void execute () {
      wait IEv.TempData(in float dat) {...}
           IEv.TempLow(in float dat) {...}
           IEv.TempHigh(in float dat) {...}
    }
  }
  myTemp tp;
}

Component implementation Light {
  Active myLight {
    void execute () {
      wait IEv.LightData(in float dat) {...}
           IEv.LightLow(in float dat) {...}
           IEv.LightHigh(in float dat) {...}
    }
  }
  myLight lp;
}
```

Finally, the actor component implementation is very simple. It is only necessary to include the components that interact with sensors. Additionally, it is necessary to connect the `Actor` interfaces with the included components, `c1` and `t1`. It is important to indicate that the reusability of components is one of the main characteristics, that is, when the sensor-actor components are specified, developers only need to include in their applications:

```
Component Actor {
    interface ITemperature {
      void TempData(in float dat);
      void TempLow(in float dat);
      void TempHigh(in float dat);
    }

    interface Light {
      void LightData(in float dat);
      void LightLow(in float dat);
      void LightHigh(in float dat);
    }


Component implementation Actor {
    Temperature t1;
    Light c1;

    input ITemperature <-> t1.IEv;
    input ILight <-> c1.IEv;

    void execute () {
       actions with light or
       temperature data
    }
}
```

In the example below, sensors get the light and temperature information and they send via the output interfaces `ITemperature` and `ILight` invoking the respective operations:

```
component Mote {
 output ITemperature with oevt;
 output ILight with oevl;
}

component implementation Mote {
 Active MoteReader {
    void execute () {
```

```
        GetData(x); //ADC
        call oevt.TempData(x.Temp);
        call oevl.LightData(x.Light);
    }
} }
```

UM-RTCOM allows us to define a priority order and we would be able to establish an order to accept the temperature event before the light event when an actor receives information simultaneously. Section III-C presents a special component to assign these QoS issues to actors and sensors.

### *B. Actor Interaction Components*

Middleware for the sensors can include components for the interaction with actors, i. e. components can include simple operations requested by actors. In the example below, the component provides methods to actors to obtain the average, maximum or minimum temperature of `number` samplings. The methods can be executed in the sensor and this reduces the communication of data between sensor and actor. If the average calculation was carried out in the actor, data must be communicated by sensors which increases the network load.

```
Component Temperature {
 interface IEv {
    void AverageTemp(out float dat, in int number);
    void MaxTemp(out float dat,in int number);
    void MinTemp(out float dat,in int number);
 }
 }
```

As commented in section III, a `connected` primitive can be used by actors to select the sensors where they can get information.

### *C. The QoS Component*

The QoS component allows developers to define the component behavior meeting the quality of service requirements of the application. The primitives that we propose in this paper are related to the establishment of the component timing requirements.

As stated before, the component model allows us to define the execution period of components. This assignment is static and it is done when an instance of the component is created.

```
MoteReader mt with period 10;
```

Mote `mt` will execute method `execute` with a period of 10 ms.

As commented in section III, an actor can receive different event types from different sensors, so a mechanism to deal with these events has to be defined to guarantee that the most important events are attended to before non critical ones. The `priority` primitive allows us to define the priority of the component methods. In the example below, a priority of level 10 is assigned to method `TempData` of the component `Temperature` and a priority of 9 is assigned to method `LightData` of the component `Light`

```
Temperature constraints TempData priority 10;

Light constraints LightData priority 9;
```

Both primitives, `priority` and `period`, define real time characteristics in a static way but it may be necessary to change these real time requirements dynamically to change the data communication sensor-actor, for example, if a network overhead exists. In this sense, every actor/sensor incorporates a special component called `QoS` that allows us to change real time requirements dynamically.

```
Component QoS {
  interface IQos {
  void Period(IComponent theComponent,in string
              interface,in string method,
              in short value);
  void Priority(IComponent theComponent,
              in string interface,
              in string method, in short value);
  short BatteryLevel();
  bool Status();
  }
}
```

The `Period` method will be invoked by actors to control the network overhead. For example, actors need deal with different data from sensors and the period of received data may be faster than how they work. In this case, an actor will call the method to change the period of the sensor data sending. Additionally, it allows sensors to reduce their activity thereby improving the battery consumption. The parameters include the component, the interface and the method where the new period will be changed.

`BatteryLevel` and `Status` Methods will be used by actors to get the battery level and the status, active or inactive, of sensors or actors. `Status` can be used to configure the network and `BatteryLevel` can be used to define an acting scheme for the actors depending on the battery charge level. `Priority` will be used when the temporal requirements are not met and a priority change in the methods is necessary.

From a low level implementation point of view, as we commented, each component has a queue associated and its implementation must be based on the priority of the invoked method in order to attend to the most critical event. Additionally, the scheduler can activate a mechanism to delete old events from the queue. This improves the QoS of the network because when several invocations of the same event are stored in the queue it is only necessary to attend to the last event received.

### *D. Actor Coordination Component*

Another important issue for WSANs is the actor coordination [4]. When WSANs are multi-actors, tasks to be carried out must be delivered among the necessary actors. Although actor selection can be done in a centralized or distributed way, in this paper we only consider the centralized decision process. The actor coordination component provides a set of primitives to carry out the task assignment.

In a centralized decision, an actor (i.e., a component of this actor) will send an `announcement message` related to the event and task. Based on the feedback from the actors, it will select and assign the task. MWSAN offers a new component, `ActorCoordination` to carry out the coordination among actors. This component includes a method


Fig. 6. Protocol for a centralized decision

called `StartAnnouncement` to send information to the neighbor actors about the event to control. Its implementation will raise the `EventAnnouncement` event. Method `ResponseAnnouncement` will receive data from actors to define the tasks to assign to the different actors. Method `DevelopAction` will assign the task to the most appropriate actors raising event `DoAction`. The sequence diagram in figure 6 shows the basic protocol for the centralized decision of multi-actor tasks. We suppose `n` actors and `Actor1` is responsible for the task assignment decision.

Methods `Status` and `BatteryLevel` belonging to the `QoS` component can be used for the actor coordination in order to decide the task assignment.

```
Component ActorCoordination {
  interface IAc {
    void StartAnnouncement (in Event ev)
    void ReponseAnnouncement(in stream data);
    void DevelopAction(in Task tk);
  }
  publishes EventAnnouncement(in Event ev),
            DoAction (in Task tk);
  consumes EventAnnouncement(in Event ev),
           DoAction (in Task tk);
}
```

## IV. Implementation

In this section we describe the details relating to the design and implementation of a prototype for the proposed middleware that we are currently working on. In this paper we use an implementation of RT Java called jRate for actors. Java offers important characteristics such as platform independence, exception handling and simplified memory management but this latter issue, the garbage collector, reduces the predictability of the real time applications. The real time specification of Java reduces garbage collection, introducing new types of memory regions and real time threads. Sensors are Crossbow Micaz type [20] and the applications are executed on Tiny OS. For this sensor type, MWSAN has been specified in UM-RTCOM but its implementation is in nesC. The implementation could change depending on the sensor characteristics.

In our proposal, the developer uses the elements provided by the middleware and UM-RTCOM to develop the applications (generic, active, passive, event, etc). Later, the environment tools will map these elements to elements of RT Java.

Each generic component is transformed into a java class exposing the input interfaces together with all the elements encapsulated in the generic component.

Active components are mapped into `RealtimeThreads` of RT Java together with the private data members and methods of the active. On the other hand, passive components are mapped to RT Java classes with the methods and data members that the passive provides. We use RT Java control mechanisms to avoid priority inversion in the access to passive components.

The code below represents part of the implementation of the QoS component described in section III-C and its using by an Actor component. A new class `QoS` is created together with a new Java interface called `IQoS` which allows to use the services of the `QoS` component. The resulting class has additional methods such as `createInstance` to create new instances of the component. The actor component creates a new instance of `QoS` and uses it invoking the `Priority` service to change the priority of method `TempHigh`. The arguments, data types, etc. are mapped to native Java data types. More importantly, all these classes are transparent for the user who only uses UM-RTCOM and MWSAN.

```
interface IQoS {
    void Period(IComponent theComponent,
      String interface,String method,short value);
    void Priority(IComponent theComponent,
      String interface,String method,short value);
    short BatteryLevel();
    boolean Status();
};

class QoS implements IComponent {
    void createInstance(String name) {...}
    IQoS getIQoS() {...}
}

class Actor implements IComponent {
    private QoS qs;

    void createInstance(String name) {
      qs=new QoS();
      qs.createInstance("qos1");
    }

    void execute() {
      IQoS iqos;
      iqos=qs.getIQoS();
      iqos.Period(this,"ITemperature","TempHigh",250);
    }
```

## V. Example

This section sketches one of the initial applications that has been considered to test the middleware. The application aims to control the temperature inside the four buildings constituting our Faculty of Computer Science at the University of Málaga. There is and actor per building. The motes inside a building send the sensed data to the corresponding actor which must maintain a constant temperature for building. In order to apply the actor coordination component a global temperature for the faculty can be carried out.

The following code shows component `ActorBuilding`. It includes components `QoS`, `ActorCoordination` and `Temperature`. The implementation gets the temperature from the sensors and makes use of component `ActorCoordination` to calculate the building temperature.

```
component ActorBuilding {
  input ISA;    // Interface sensor-actor
  input IAc;    // Interface Actor coordination
  output IAc with osa;  // Interface Actor coordination
}

component implementation ActorBuilding {
    QoS qs;
    Temperature temp;
    ActorCoordination ac;
    // Delegate services
    input ISA <-> temp.ISA;
    input IAc <-> ac.IAc;

    void execute() {
        wait ISA.GetTemp(temp);
        -- Temperature Control
        call osa.StartAnnouncement("Temperature");
        wait IAc.ResponseAnnouncement(vl);
        -- Calculate the average temperature
        call osa.DoAction(newval);
    }
 }
```

As commented in section IV, UM-RTCOM is implemented in RT JAVA. The following code represents part of the implementation of the `ActorBuilding` component.

```
class ActorBuilding extends RealtimeThread implements
IComponent {
    private QoS qs;
    private Temperature temp;
    private ActorCoordination ac;
    private float temp;
    private IAc iac,osa;
    private ISA isa;

    void createInstance(String name) {
      //Initializations
    }
    ISA getISA() { return isa; }
    IAc getIAc() { return iac; }

    private void init() {
    // Delegate services
        isa.connect(temp.getISA());
        iac.connect(ac.getIAc());
    }
    public void run() {
        isa.wait("GetTemp");
        -- Temperature Control
        osa.call("StartAnnouncement");
        isa.wait("ResponseAnnouncement");
        -- Calculate the average temperature
        osa.call("DevelopAction");
    }
 }
```

## VI. Conclusions

This paper presents the MWSAN middleware that provides a set of high level services to build WSAN applications. MWSAN focuses on the requirements of the QoS and the actor coordination present in this kind of system. MWSAN has been specified in UM-RTCOM, a component language that provides constructions to define real time characteristics, such as for example, a priority schema where the highest priority events are executed first. These issues improve the temporal behavior of WSAN applications. The model implementation for actors has been carried out in jRate, an RT JAVA implementation in order to meet the real time requirements that MWSAN requires.

As future work, we are currently defining new components for MWSAN to provide high level services related to routing or system monitoring. In this paper we have developed a prototype to map UM-RTCOM to RT Java but a complete automatic tool will be implemented in the future.

## 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] Blumenthal J., Handy M., Golatowski F., Haase M., Timmermann D., Wireless Sensor Networks - New Challenges in Software Engineering. *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, 2003. IEEE Computer Society Press.

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

[7] Diaz M., Garrido D.,Llopis L., Rus F., Troya J.M. Integrating Real-Time Analysis in a Component Model for Embedded Systems. *Proceedings of the 30th IEEE Euromicro Conference. 2004*

[8] Corsaro A.,Schmidt D.C., The design and Performance of the jRate Real-Time Java Implementation. *On the Move to Meaningful Internet Systems 2002*, pp. 900–921, Lecture Notes in Computer Science (2002).

[9] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D., The nesC Language: A Holistic Approach to Networked Embedded Systems. *Proceedings of Programming Language Design and Implementation (PLDI 2003)*, 2003.

[10] Tiny OS Inc. http://www.tinyos.org/

[11] Boussinot F. and Simone R. The ESTEREL language *Proceedings of the IEEE, 79(9):1293-1403.Sept. 1991.*

[12] Benveniste A., Guernic P.L. and Jacquemot C. Synchronous programming with events and relations: The SIGNAL language and its semantics. *Science of Computer Porgramming, 16(2):103-149,1991.*

[13] Halbwachs N., Caspi P., Raymond P. and Pilaud D. The Synchronous data-flow progamming language LUSTRE.*Proceedings of the IEEE, 79(9):1305-1320.Sept. 1991.*

[14] Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P., TinyLime: Bridging Mobile and Sensor Networks through Middleware. *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communication (PerCom 2005)*, pp. 61–72, Kauai Island, Hawaii, 2005. IEEE Computer Society Press.

[15] Fok, C-L., Rooman, G-C.,Lu, C., Rapid Development and flexible deployment of adaptative wireless sensor network applications. *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pp. 135–151, Columbus, USA, 6–10 June, 2005. IEEE Computer Society Press.

[16] Shen C-C., Srisathapornphat C., Jaikaeo C., Sensor Information Networking Architecture and Applications. *IEEE Personal Communications*, (2001) pp. 52–59.

[17] Liu, T., Martonosi M., Impala: A Middleware System for Managing Automomic Parallel Sensor Systems. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 107–118, San Diego, California, USA, 2003.

[18] Díaz M., Garrido D., Llopis L., Rubio B., Troya J.M., A Component Framework for Wireless Sensor and Actor Networks. *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA06)*, pp. 300–308 IEEE Computer Society Press.

[19] Bollela, Gosling et al. The Real Time Specification for Java. *Addison-Wesley, 2003*

[20] Crossbow Technology Inc. http://www.xbow.com/