A Component Framework for Wireless Sensor and Actor Networks

Manuel Díaz, Daniel Garrido, Luis Llopis, Bartolomé Rubio, José M. Troya
Depart. Languages and Computing Science
University of Málaga
{mdr, dgarrido, luisll, tolo, troya}@lcc.uma.es

Abstract

Wireless Sensor and Actor Networks (WSANs) constitute an emerging and pervasive technology that is attracting increased interest for a wide range of applications. WSANs have two major requirements: coordination mechanisms for both sensor-actor and actor-actor interactions, and real-time communication to perform correct and timely actions. Additionally, the development of WSAN applications is notoriously difficult, due to the extreme resource limitations of nodes. This paper introduces a framework to facilitate the task of the application programmer taking into account these special characteristics of WSANs. We propose a real-time component model using light-weight components. In addition, a high-level coordination model based on tuple channels is integrated into the framework including high-level constructs that abstract the details of communication and facilitate the data-centric behavior of sensor queries.

1. Introduction[1]

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 various application areas, such as environmental monitoring, object and event detection, military surveillance, precision agriculture [2][3].

One variation of WSNs that is rapidly attracting 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 powers 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.

A major challenge in WSNs that becomes even more important in WSANs is the coordination requirement [5][4]. In WSANs two kinds of coordination, sensor-actor and actor-actor coordination, have to be taken into account. In particular, sensor-actor coordination provides the transmission of sensed data from sensors to actors. An actor may need every reading from its area sensors or may be interested in receiving only some information (dealing with typical queries in sensor networks such as "which region has a temperature higher than 30ºC?"). After receiving sensed data, actors need to coordinate with each other in order to make decisions on the most appropriate way to perform the action. It must be decided whether the action requires exactly one actor (and which one) or, on the contrary, if it requires the combined effort of multiple actors. On the other hand, depending on the application there may be a need to respond rapidly to sensor input. Moreover, the collected and delivered sensor data must still be valid at the time of acting. Therefore, the issue of real-time is a very important requirement in WSANs. Thus, coordination models and communication protocols should support real-time properties of this kind of system.

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.

The component-oriented paradigm seems to be a good alternative [6]. Software components [7] offer several features (reusability, adaptability, ...) very suitable for

1. This paper has been funded in part by EU project FP6 IST-5-033563 and Spanish project TIN2005-09405-C02-01

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 (CCM, COM, JavaBeans) either suffer from extensive resource demands (memory, communication bandwidth, ...) or dependencies on the operating system, protocol or middleware. In this paper we present a component framework for WSANs that consists of two elements: a real-time component model and a high-level coordination model. UM-RTCOM [8], a previous development focused on software components for real-time distributed systems, is 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.

On the other hand, TC-WSANs, a high-level coordination model based on TCMote [9][10], is integrated into the framework in order to satisfy the above mentioned coordination requirements. 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.

Some approaches have incorporated the component-oriented paradigm to deal with WSAN programming. The most used is possibly nesC [11]. It is an event-driven component-oriented programming language for networked embedded systems. However, 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 allows real-time analysis to be performed. Other languages such as Esterel [12] and Signal [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]. However, since the above listed requirements impose stricter constraints, they may not be suitable for application to WSANs.

Moreover, although there has been some research done related to WSANs [17], to the best of our knowledge, none of the existing work to date proposes coordination models which provide high-level constructs to ease the task of the programmer and which provide the unique features and requirements of WSANs.

The rest of the paper is structured as follows. In Section 2 the reference operational setting is described. Section 3 presents the UM-RTCOM model, including its main elements, syntax and programming style. Section 4 presents the framework including the coordination model TC-WSANs with its main design goals, concepts and primitives. Section 5 shows an example where our framework is used. Finally, some conclusions and future work are sketched in Section 6.

2. 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 only send 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. 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).

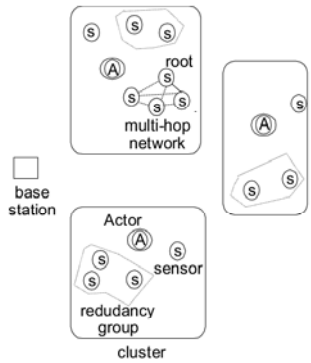


Figure 1. Operational setting.

Clustering avoids the typical situation in WSANs where multiple actors can receive information from sensors about the sensed phenomenon. 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.

3. 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 the main features of this model have motivated 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 like executable or libraries with a minimum overhead. In addition, UM-RTCOM components are complemented by an abstract model of their behaviour (based on SDL [18]). This abstract model allows us to perform different analysis types such as for example real-time analysis, deadlock freedom, liveness properties, etc. In this sense, a UM-RTCOM component is the sum of the code and abstract model.

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.

3.1. Component types

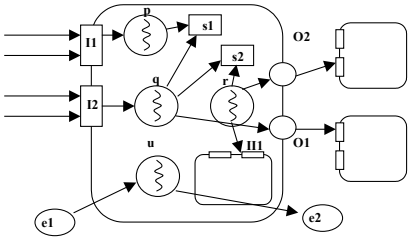
There are two main component types: primitive and generic. Generic components are the standard components of the model. They provide services through interfaces and can require services of other generic components. On the other hand, primitive components (active or passive) are contained in generic components.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

Generic Components: These components are the basis of the model. They act like 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 what events produce and what events consume.

This way, components can communicate with each other without common interfaces.




```
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>);
```

Figure 2. Generic component.

Fig. 2 shows an example of a generic component with the declaration of input and output interfaces, events consumed and produced and the primitive components included.

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 analysis phases.

Active components are responsible for the execution flow inside generic components through the interaction with other elements such as passive components 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 utilization 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.

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

behaviour is used in order to facilitate later analysis phases. Passive components provide mutual exclusion with priority ceiling mechanisms which avoid priority inversions.

3.2. 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.

Wait Primitive: This primitive is used to waiting for new invocations on services or the creation of consumed events. It is used in Active components and its syntax is the following:

```
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 indicated to avoid infinite waiting.

Call Primitive: Primitive call is used to invoke services of generic components. It can be used in Active or Passive components and its syntax is the following:

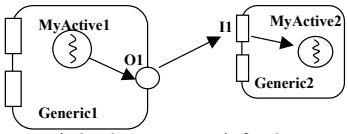
```
call <ref>[.<interf_name>].<method>(
      <parameters>) [Time <time>
```

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

Raise Primitive: This primitive is used to raise events in an asynchronous way.

When a component raises an event, this event is caught by all the components that consume that event. Its syntax is the following:

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




```
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);
    }
}
```

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

Figure 3. Synchronization primitives.

Fig. 3 shows an example where two generic components (`Generic1` and `Generic2`) are connected. 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.

3.3. 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.

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.

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 elements visible 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 is the following:

```
instance constraints <interf_name.method>
      Period P,Deadline T ';'
constraints <event_type> Deadline D, Period P ';'
```

Real-time analysis: UM-RTCOM allows real-time analysis of full systems. This analysis is based on three elements: the abstract model, the utilization of the synchronization primitives and annotations on the code.

The model only allows communication between components using the synchronization primitives. This is important because this way, we can divide the code into different parts which can be analyzed.

So, the synchronization primitives act like division points and the code of the active components can be expressed like sequential blocks between synchronization primitives.

The analysis model is based on the utilization of the abstract models of the components which use a real-time extension of SDL presented in [8]. The model is complemented with annotations on the source code of the components to allow an estimate of the worst case execution times (WCETs) on the final platform.

Initially, the special features of WSAN systems do not allow us to use this analysis technique on the whole system (i.e. sensors), but we think some analysis can be

applied to actors as they are components with higher processing capacities. This is an important innovation in comparison with other approaches [11].

4. The framework for WSANs

UM-RTCOM allows us to define components with a functionality and then interconnect them to get an application. As stated before, however, WSANs need to incorporate a coordination mechanism to define the interaction among nodes (actors and sensors) and this mechanism can be viewed as a middleware between the applications and the operating system offering a set of services.

In this sense we can use UM-RTCOM to define a special component offering these services making a framework that takes into account the real-time and coordination characteristics of this kind of system.

In the following section we describe the coordination model and encapsulate it in a UM-RTCOM component to be used in the application composition.

4.1. The TC-WSANs model

A coordination model can be viewed as a triple (E,M,L), where E represents the entities being coordinated (agents, objects, processes, ...), M is the media used to coordinate the entities (shared variables, channels, tuple spaces, ...), and L is the semantics framework the model adheres to (guards, associative access, synchronization constraints, ...). It is embedded in a programming (base or host) language to develop parallel and distributed applications.

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, i.e. actors and sensors (only actors in a super-cluster). 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 (between the leader actor and the rest of the actors in a super-cluster).

Finally, the coordination "laws" L that govern the actions related to coordination are determined by the semantics of every model primitive (storing and withdrawing of channels, asynchronous sending of tuples through a channel, blocking consumption of tuples from a channel, ...).

4.2. Tuple Channel Space

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);
   void react(in string operation,in string reaction);
   void find_r(in channel_attributes
         search_pattern, out short chanid);
   void no_r(in channel_attributes
           search_pattern,out bool ok);
   // channel interconnection
   void interconnect(in short chanid_target,
         in short chanid_source);
   void send(in short chanid_target,
         in short chanid_source);
   void connector_instance(in
         lchannel_id ch_in_list,in short ch_out,
         in string behaviour);
   // 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, not only the sensors and the actors but also 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, consider a cluster split into four quadrants. It could be useful to assign different channels to quadrants. Channels can be provided with an attribute such as `location`. So, for a channel assigned to the southeast quadrant, the attribute-based data structure identifying it could be the following:

```
[com_type = one_to_many, location = S_E]
```

One of the characteristics that contributes to achieving the real-time requirements demanded by WSANs is the priority issue.

Some activities are more important than others and should be scheduled in an appropriate way in order to enhance the system response time. In our approach, 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. For example, an actor can check the priority of different channels in order to establish the order in which their data should be obtained.

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. For example, the following operation:

```
destroy([com_type = ?, location = S_E])
```

will withdraw from the TCS all channels assigned to the southeast quadrant.

A search_pattern is also used by the find primitive in order to find an appropriate channel to establish some communication. The find primitive is governed by a pattern matching scheme where, if several candidates are found, one of them is chosen in a non-deterministic way.

A reaction can be associated to a TCS operation by means of the react primitive, which has two arguments: the implied operation and the desired reaction associated with it. The occurrence of the operation will trigger the reaction. A reaction is defined as a conjunction of non-blocking operations, and has a success/failure transactional semantics: a successful reaction may atomically produce effects on the TCS, a failed reaction yields no result at all. The operations we have initially considered as candidates to be included inside a reaction are create, get_attr, destroy, find_r and no_r. find_r is similar to find, but it is a non-blocking primitive, i.e. it immediately returns a null value if no channel is found. Complementary to it, the primitive no_r succeeds (a true value is returned) when its argument does not match any data structure in the TCS, but fails otherwise, returning a false value.

For example, consider that a sensor uses the find primitive in order to find a channel to interact with the cluster actor. If there is no channel matching the search pattern, the primitive will block and the sensor will receive no channel. It is highly probable that the actor does not want this situation to occur. This way, it can previously associate the following reaction to a find operation:

```
react(find([com_type = one_to_many, location = S_E]),
   (no_r([com_type = one_to_many,location = S_E]),
   create([com_type = one_to_many, location = S_E])))
```

After the execution of the find operation, the specified reaction is triggered. If no channel has been found, no_r operation succeeds and create creates a new channel in the TCS. This will resume the find operation, which will provide the sensor with the required channel.

Reactions can also be associated to operations performed inside reactions. This way, an operation may in principle trigger a multiplicity of reactions. However, all the reactions executed as a consequence of a TCS operation are all carried out in a single transition of this space, before any other operation is served.

Primitives connect, put, get and disconnect will allow us to operate with the channels. These operations internally interconnect with component CTupleChannel described below.

4.3. Tuple Channel

A Tuple Channel is a priority queue structure that allows both one-to-many (from an actor to some sensors belonging to its cluster or from a cluster leader actor to some actors belonging to its super-cluster) and many-to-one (vice versa) communication schemes. Both communication schemes are carried out in a single-hop way. However, our proposal does not preclude the existence of a multi-hop sub-network, but the multi-hop mechanism is independent of the TC-WSANs model.

Here, the priority issue affects tuples, i.e. data structures communicated through channels, as is explained below. 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. On the other hand, when it no longer needs a channel, it executes the disconnect primitive in order to disable the TC connection. Connection information will be useful to the run-time system in order to achieve an efficient channel implementation.

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 depends on the specified attributes. The possible attributes that we have initially considered to be associated with a tuple are the following:

- priority. It establishes the priority of the tuple. Tuples are ordered inside the channel by priority. Tuples with higher priority are the first obtained by the consumers. This allows an actor to attend to the highest priority messages or events first.
- deadline. It establishes the maximum time the tuple is going to stay in the channel. Besides achieving real-time requirements, the garbage collection carried out by the run-time system will be improved by means of this attribute.
- remove. It indicates whether older tuples of the same kind should be removed from the channel before adding the new one. In some cases, only the most current reading from a sensor is needed by an actor. This attribute precludes the actor from getting old readings from the same sensor.

A consumer will use the get primitive to receive information from a channel. Each time a get operation is executed, a new tuple from the beginning of the channel

is obtained. When a channel consumer obtains a tuple, this is not withdrawn from the channel from the view of the other consumers and so, the same information can be received by all of them. We can say that each consumer accessing a channel will have, at any time, a view of it, which may be different from the views of the rest of the consumers sharing the channel. Fig. 4 shows an example describing this consumption behaviour. There are 3 consumers sharing the channel TC.

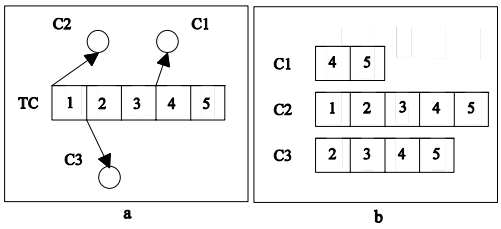


Figure 4. Consumption behaviour.

Fig. 4.a shows a situation where consumer C1 has consumed the first three tuples, C3 has only consumed the first one, and C2 has consumed none.

Fig. 4.b represents the view that each consumer has of the channel TC. This consumption behaviour contributes to dealing with the data-centric characteristics of sensor queries.

Both one-to-many and many-to-one communication schemes are appropriate for dealing with typical queries in sensor networks such as "which region has a temperature higher than 30ºC?". The consumption behaviour just described allows every region to receive the same query from a channel in an one-to-many way, as shown in Fig. 5.a, where consumers C1, C2 and C3 receive the same query sent by the producer P through the channel TC1. Then, implied motes, for example C1 and C2 in Fig. 5.b, can answer it in a many-to-one way by means of a new channel whose identifier, TC2 in the figure, was sent inside the tuple representing the query.

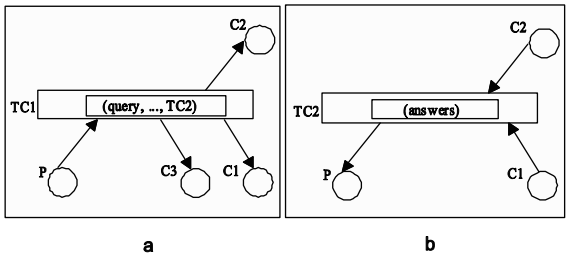


Figure 5. Communication schemes.

4.4. Channel interconnection

Channels can be dynamically interconnected. The model incorporates this possibility by means of two kinds of channel connectors: the user-defined connectors and the predefined ones. The predefined connectors are `interconnect` and `send`. The former sends the data contained in channel `chanid_source` to `chanid_target`. After the interconnection, any data sent through `chanid_source` will also be sent through `chanid_target`. The latter acts in the same way but the channels remain independent after their interconnection.

A user-defined connector is a coordination structure that takes as arguments a set of input channels and a single output channel. The user specifies the behaviour of a channel connector type, i.e. how the input tuples (coming from input channels) are selected and sent through the output channel. Different instances of the same connector type can be created by means of the `connector_instance` primitive.

Different benefits can be achieved in WSAN applications by using dynamic channel interconnection. For example, in our operational setting, an actor can redirect data received from its sensors to the cluster leader actor (hierarchical structure) by interconnecting the corresponding channels. On the other hand, data aggregation can be carried out in an elegant way and it is also possible to eliminate some redundant data by specifying the appropriate connector type. Finally, based on the different priorities of the input channels, the appropriate prioritized scheme can be established in order to obtain the tuples from them.

5. Example

This section sketches one of the initial applications that have been considered to test the framework. The application tries to control the temperature inside the four buildings constituting our Faculty of Computer Science at the University of Málaga. The motes inside the buildings send the sensed data to the actor which must maintain a constant temperature for each building.

The following code shows the `Actor` component. This component creates the channels (one per building) and defines their attributes (location,...). It is composed of 4 active components (`act1,...,act4`) to get data from the four buildings. The `Actor_Location` components periodically read from the channel associated to their respective buildings (by using `tcs`) to get data from the sensors.

```
component Actor {
  output ITupleChannelSpace with tcs;
}
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() {
      tuple t;
      call tcs.get(chanid_Actor,INFINITE,&t);
     // Do control
   }}
  Actor_Location act1("1_Building"),
      act2("2_Building"),act3("3_Building"),
```

```
      act4("4_Building") with period 100;
}
```

The components `Mote` get the channel associated to their location (`find`) and periodically send the sensed data by using the `MoteReader` active component.

```
component Mote {
  output ITupleChannelSpace with tcs;
}
component implementation Mote {
   short chanid_Actor;
   Mote() { // Location and Connection to Actor
    channel_attributes fa; fa.length(1);
    fa[0].name="Location"
    fa[0].value=Get_Location();
    call tcs.find(fa,&chanid_Actor);
    call tcs.connect(chanid_Actor,"P");
   }
   Active MoteReader {
    void execute() {
      tuple t;  // Get data from environment
      call tcs.put(chanid_Actor,t);
   }}
   MoteReader mt with period 100;
}
```

6. Conclusions and future work

We have presented a component framework for developing WSAN applications. The reference operational setting, based on a (hierarchical) architecture of sensor clusters, each one governed by an actor has been described. The real-time component model proposed uses light-weight components which are complemented with an abstract model of their behaviour. It allows us to perform different types of analysis such as real-time, deadlock freedom and liveness properties. The node communication and synchronization mechanisms are based on tuple channels, which are priority queues that allow one-to-many and many-to-one communication schemes which together with the attribute-based naming scheme used permit the data-centric characteristics of typical sensor queries and satisfy the real-time requirements of WSANs. Channel interconnection provides great flexibility for the definition of different interaction protocols.

We are currently developing a prototype to support the proposed framework. It must be completed with the operating system for the application execution. This OS must guarantee the real-time characteristics of UM-RTCOM. Mantis OS [20] is an embedded multithreaded operating system for wireless micro sensor platforms that meets our requirements. The motes used are the Crossbow Micaz sensors [21].

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 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.

[9] 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.

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

[11] 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*, pp. 1-11.

[12] F. Boussinot, R. Simone, "The ESTEREL language", *Proc.of the IEEE*, vol. 79, no. 9, pp. 1293-1403, 1991.

[13] A. Benveniste, P.L. Guernic, C. Jacquemot, "Synchronous programming with events and relations: The SIGNAL language and its semantics". *Science of Comp. Programming*, vol. 16, no. 2, pp. 103-149, 1991.

[14] 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.

[15] C-L. Fok, G.C. Rooman, C. Lu, "Rapid Development and flexible deployment of adaptive wireless sensor network applications" in *Proc* of the 25th Int. Conference on Distributed Computing Systems, 2005, pp. 135–151.

[16] T. Liu, M. Martonosi, "Impala: A Middleware System for Managing Automomic Parallel Sensor Systems", *ACM SIGPLAN PPoPP*, 2003, pp. 107–118.

[17] 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.

[18] ITU recommendation Z.100, SDL, 2000.

[19] F. Silva F., J. Heidemann, R. Govindan, D. Estrin, "Directed Diffusion. In Iyengar", S. and Brooks, Frontiers in Distrib. Sensor Networks, CRC Press, 2004.

[20] 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.

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