Programming Wireless Sensor and Actor Networks with TC-WSANs

Javier Barbarán, Manuel Díaz, Iñaki Esteve, Daniel Garrido, Luis Llopis,
Bartolomé Rubio and José M. Troya
Dpto. Lenguajes y Ciencias de la Computación. Málaga University
29071 Málaga, SPAIN
Corresponding author: Bartolomé Rubio (tolo@lcc.uma.es)

***Abstract*—Wireless sensor and actor networks (WSANs) constitute a new pervasive technology and currently one of the most interesting fields of research. WSANs have two major challenges: coordination mechanisms for both sensor-actor and actor-actor interactions, and real-time communication to perform correct and timely actions. This paper introduces a middleware architecture that address these two requirements and facilitate the application programmer task. Our proposal supports a high-level coordination model that is based on the use of tuple channels to achieve communication and synchronization among the nodes. A tuple channel is a priority queue structure that allows data structures to be communicated both in a one-to-many and many-to-one way, facilitating the data-centric behavior of sensor queries. The main characteristics of the model and the different components forming the middleware are presented. In addition, some implementation details and preliminary results of the current prototype are described.**

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 [2]. WSNs have been successfully used for various application areas, such as environmental monitoring, object and event detection, military surveillance, precision agriculture [13].

A variation of WSNs that is rapidly attracting interest among researchers and practitioners is the so called *Wireless Sensor and Actor Networks* (WSANs) [1]. 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 [10] [1]. 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. 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 has to be decided whether the action requires exactly one actor (and which one) or, on the contrary, 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 real-time issue is a very important requirement in WSANs. Thus, coordination models and communication protocols should support real-time properties of this kind of system.

This paper presents a middleware architecture to facilitate application development over WSANs. Our reference operational setting is based on a scenario 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. Several actors may form a (super-)cluster which is governed by one of them, the so called cluster leader actor. Clustering avoids the typical situation in WSANs where multiple actors can receive the information from sensors about the sensed phenomenon. The lack of coordination between the sensors may cause too many and unnecessary actors to be activated and as a result the total energy consumption of all sensors can become high. Moreover, clustering together with the single-hop communication scheme minimize the event transmission time from sensors to actors, which contributes to support the real-time communication required in WSANs.

The proposed middleware supports TC-WSANs [3], a high-level coordination model that uses *Tuple Channels* (TCs)

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

to both sensor-actor and actor-actor interactions. A TC is a priority queue structure that allows one-to-many and many-to-one communication of data structures, represented by tuples. The use of TCs makes possible the realization of a requirement for communication in WSANs referred to as the ordered delivery of information collected by the sensors [1]. In addition, the channel consumption behaviour proposed contributes to dealing with the data-centric characteristics of sensor queries. The priority issue, taken into account at two levels, channels and tuples, contributes to achieving the real-time requirements of WSANs. In the middleware architecture proposed, each actor owns a *Tuple Channel Space* component, which stores the tuple channels used to carry out communication and synchronization between the actor and the sensors/actors belonging to the cluster governed/leadered by it. TCs can be dynamically interconnected through the use of predefined and user-defined connectors, providing great flexibility for the definition of different topologies. Sensor data dissemination can be achieved in an elegant way, allowing for data redirection, data aggregation and redundant data elimination.

This work is being carried out in the context of the SMEPP (Secure Middleware for Embedded Peer-to-Peer Systems) project [19], a recently initiated EU funded project (FP6 IST-5-033563), which has the general goal of developing a new secure, generic and highly customizable middleware, based on a new network centric abstract model for EP2P systems.

The rest of the paper is structured as follows. To conclude this introduction, we present some related work. Section II gives an overview of the main characteristics of the TC-WSANs model. The middleware architecture is presented in Section III. In Section IV some implementation details and preliminary results of the current prototype we are developing are given. Finally, some conclusions are sketched in Section V.

A. Related Work

The coordination needs in WSNs and WSANs have attracted the attention of the Coordination paradigm community [4]. More concretely, different coordination models and middleware based on Linda abstract model [14] have appeared. Linda can be considered the most representative coordination language. It is based on a shared memory model where data is represented by elementary data structures called tuples, and the memory is a multiset of tuples called a tuple space. The flexibility of this coordination model is demonstrated by the fact that, even if it has been designed more than twenty years ago for programming parallel computers, it is still considered a referring model in the rather different context of distributed programming. In [11], the mobile agent based middleware Agilla is presented. Agilla allows users to create and inject special programs called mobile agents that coordinate through local tuple spaces, and migrate across a WSN performing application-specific tasks. This fluidity of code and state has the potential to transform a WSN into a shared, general-purpose computing platform capable of running several autonomous applications at a time.

In TinyLime [8], a new operational scenario is assumed, one that naturally provides contextual information, does not require multi-hop communication among sensors, and places reasonable computation and communication demands on the motes. Sensors are sparsely distributed in an environment, not necessarily able to communicate with each other, and a set of mobile base stations (laptops) move through space accessing the data of sensors nearby. Each base station owns a tuple space and federated tuple spaces can be established in order to communicate and synchronize several base stations and some clients hosts. TeenyLime [6] is an evolution of TinyLime to afford the aforementioned unique features and requirements of WSANs. As in TinyLime, the core abstraction is the transiently shared tuple space. However, unlike TinyLime the spaces are physically located on the devices themselves.

Our approach however is based on the use of tuple channels instead of tuple spaces to carry out communication and synchronization among the involved nodes. Several advantages can be obtained from the use of channels with respect to shared memory models:

- *Architectural expressiveness*. Like messaging, using channels to express the communication carried out within a distributed system is architecturally much more expressive than using shared data spaces. With a shared data space, it is difficult to see which components exchange data with each other.
- Channels ease *hierarchical clustering*. To facilitate scalable operations within sensor and actor networks, nodes should be aggregated to form clusters, based on their power levels and proximity. The aggregation process could also be recursively applied to form a hierarchy of clusters.
- Channels support *data streams in a natural and suitable way*. The application programmer does not have to deal with head and tail tuples as is necessary in a tuple space based approach to implement information streams. This is particularly important in information-flow applications, such as building sensor and actor networks.
- Channel interconnection provides great flexibility for the definition of *complex and dynamic interaction protocols*. Sensor data dissemination can be achieved in an elegant manner, allowing for data redirection, data aggregation and redundant data elimination.

Finally, a recent approach that is not based on the Coordination paradigm can be found in [5]. Instead of considering the physical nodes, it introduces the concept of virtual nodes, a programming abstraction to simplifying the development of decentralized WSAN applications. The set of nodes being virtualized is declaratively specified using the notion of logical neighborhoods [16]. This work cannot be actually considered a middleware but a programming abstraction conceived to be a simple extension to existing WSAN programming languages (e.g. nesC). Moreover, it does not directly deal with the real-time requirement of WSANs.

II. The TC-WSANs model

We propose an operational scenario where the sensors are deployed in fixed locations in the established region to sense the phenomenon. Sensors are organized in different clusters,

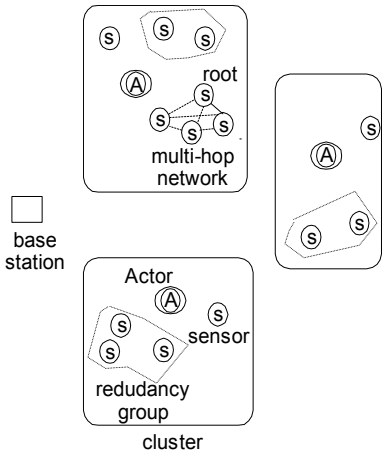

Fig. 1. Operational setting.

each one governed and controlled by an actor (Fig. 1). Unlike sensors, actors can move (inside their clusters or even to other ones) in order to carry out their actions in the most appropriate way. One example of a mobile actor is a robot, which, as could be required in some applications, may also incorporate a sensor unit.

Due to the sheer number of sensors involved in a WSAN, some degree of redundancy can be expected, improving reliability. In our operational setting, 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.

Although our approach proposes a single-hop communication between the sensors and the actor inside a cluster, it 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. So, this root sensor can send not only its measurements to the actor but also the information collected from the multi-hop sub-network.

A hierarchical structure may be achieved clustering different clusters into a (super-)cluster, whose member nodes are the corresponding actors, one of which will act as the leader. In our operational setting, the base station, which monitors the overall network and communicates with actors, can be considered as the leader actor of a cluster grouping the leader actors of outer clusters. This way, we achieve an *Automated Architecture* where sensors detecting a phenomenon transmit their readings to the actor nodes, which process all incoming data and can initiate appropriate actions, instead of having to route data to the base station (*Semi-Automated Architecture*). As sensed information is conveyed from sensors to (close) actors, latency is minimized in the Automated Architecture.

As said before, in WSANs a decision needs to be taken on the action to be performed according to the event. Our operational setting and the way communication and synchronization among nodes are carried out, imply that, although a *Decentralized Decision* mechanism can also be carried out, the most appropriate way of achieving the action decision is through a *Centralized Decision* mechanism. Actors transmit the specifications of the event such as location, intensity, etc. to the actor leading the cluster where they are, which functions as a decision center (it may be necessary to climb up in the hierarchical structure). This decision center, which has information about the actors in its cluster, selects the "best" actors for that task and triggers them to initiate the action.

A coordination model can be viewed as a triple (E,M,L), where E represents the entities being coordinated (agents, objects, processes, ...), M the media used to coordinate the entities (shared variables, channels, tuple spaces, ...), and L the coordination "laws", i.e. the protocols and rules used for coordination (guards, associative access, synchronization constraints, ...). It is embedded in a programming (base or host) language to develop parallel and distributed applications. In TC-WSANs, the entities to be coordinated E are the nodes, i.e. actors and sensors (only actors in a super-cluster). Each actor owns a Tuple Channel Space, which constitutes the coordination media M and stores the tuple channels used to carry out the communication and synchronization between the sensors and the actor inside a cluster (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 removing of channels, asynchronous sending of tuples through a channel, blocking consumption of tuples from a 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. Data structures communicated through channels are represented by 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.

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. This consumption behavior 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 $30°C$?". The consumption behavior just described allows every region to receive the same query from a channel in an one-to-many way. Then, implied motes, can answer it in a many-to-one way by means of a new channel whose identifier was sent inside the tuple representing the query.

To facilitate the data-centric characteristics of sensor queries, attribute-based naming is the scheme selected [17] [18]:

```
[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 in 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 in WSANs is the priority issue. Some activities are more important than others and should be scheduled in an appropriate way in order to enhance the system response time. In our approach, the priority issue is considered at two levels: channels and tuples.

On the one hand, 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 and establish a priority-based scheduling to get data from different channels. On the other hand, when a channel producer sends a tuple, some attributes may be associated with it. The way this tuple is treated and ordered inside the channel depends on the specified tuple 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 channel consumers. This allows an entity 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 mechanism carried out by the run-time system will be improved by means of this attribute.
- `remove`. It indicates if older tuples of the same kind should be removed from the channel before adding the new one. For example, 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.

III. The TC-WSANs Middleware Architecture

When the application programmer programs the actor and the sensors of a cluster using TC-WSANs, what s/he is actually programming is what we have called the `ActorProcess` and the `SensorAgents` inside the actor host (application layer in Fig. 2). The `ActorProcess` is in charge of doing the actor task, meanwhile a `SensorAgent` acts on behalf of a real sensor. Clearly, sensors are not really physically on the actor host, but as usual, it is the middleware that takes care of creating this abstraction to simplify the programmer's life. Therefore, the application programmer does not have to directly deal with the real sensor programming.

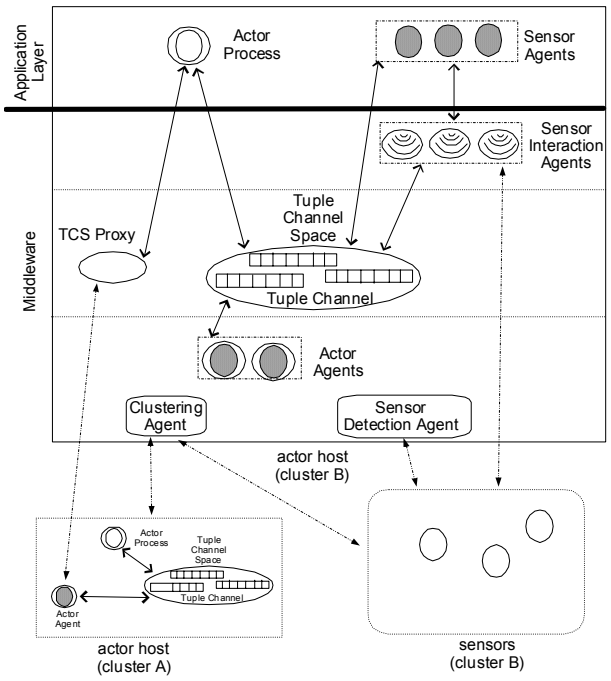

Fig. 2. Middleware architecture.

The main components of the proposed middleware deployed in both actors and sensors are described in Sections III-A and III-B, respectively.

A. On-Actor Components

In Fig. 2, the main components of the middleware architecture inside an actor host are depicted. They are presented in the following sections.

1) Tuple Channel Space: A Tuple Channel Space (TCS) is a shared data space accessed by the members of a cluster. The TCS interface provides the following primitives (together with the channel interconnection primitives that will be described later):

- `create(attributes)`. It creates a channel in the TCS. The channel identifier is returned. It is a non-blocking primitive.
- `destroy(tc_id)`. It removes the specified channel from the TCS. Nothing is done if the channel does not exist. It is a non-blocking primitive.
- `destroy(search_pattern)`. It removes one or several channels from the TCS (a matching process is carried out). Nothing is done if no channel is found. It is a non-blocking primitive.
- `find(search_pattern,time_out)`. It finds a channel in the TCS, whose identifier is returned. It is a blocking primitive, i.e. the entity invoking it will suspend its execution if no channel is found. A time-out may be established to avoid an endless wait.
- `react(operation,reaction)`. It associates a reaction to a TCS operation. It is a non-blocking primitive.

When a channel is created its attributes are specified. Besides the attributes established by the programmer, the

middleware supporting the model can establish some other system dependent attributes, such as the number of sensors connected to a channel, which can be useful in order to carry out aggregation functions.

On the other hand, when it is desirable that a channel be removed from the TCS, the `destroy` primitive is used giving as argument either a channel identifier or an attribute-based data structure (`search_pattern`). In the latter case, a pattern matching process is carried out taking into account the specified attributes. For example, the following operation:

```
destroy([location = S_E])
```

will withdraw from the TCS all channels assigned to the southeast quadrant (independently of the remaining attributes).

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`, `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 `no_r` primitive 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` operation 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.

2) Tuple Channel: Entities access a Tuple Channel (TC) by means of five primitives:

- `connect(tc_id,mode)`. The entity that executes this primitive establishes a connection to the specified channel. The entity will act as a producer (`mode = P`) or as a consumer (`mode = C`). It is a non-blocking primitive.
- `disconnect(tc_id)`. The specified channel is disconnected from the entity executing the primitive. It is a non-blocking primitive.
- `get_attrs(tc_id)`. It returns the attributes of channel `tc_id` or a null value if the channel does not exist. It is a non-blocking primitive.
- `put(tc_id,tuple,attributes)`. It sends a tuple through a channel. Some attributes may be associated with the tuple (see below). It is a non-blocking primitive.
- `get(tc_id,time_out)`. It obtains a tuple from a channel. It is a blocking primitive, i.e. the entity invoking it will suspend its execution if the channel is empty. A time-out may be established to avoid an endless wait.

Before an entity can send/receive information through/from a TC by means of the `put/get` primitive, 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.

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 the following:

- `interconnect(tc_id1,tc_id2)`. It sends the data contained in channel `tc_id2` to `tc_id1`. After the interconnection, any data sent through `tc_id2` will also be sent through `tc_id1`.
- `send(tc_id1,tc_id2)`. It sends the data contained in channel `tc_id2` to `tc_id1`. In this case, the two channels remain independent after their interconnection.

A user-defined connector is a coordination structure that takes as arguments one or more input channels and one or more output channels. The user specifies the behaviour of a channel connector, i.e. how the input tuples (coming from input channels) are selected and sent through the output channels. Different instances of the same connector can be created.

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, it is also possible to eliminate some redundant information and data aggregation can be carried out in an elegant way by specifying the appropriate connector type. For example, an actor can develop a temperature average function (to aggregate readings coming from several sensors and send the result to the base station or the cluster leader actor) by establishing a channel connector that takes one or several input channels (shared by the sensors and the actor) and one output channel (shared by the actor and the base station). Finally, based on the different priorities of the input channels (these priorities

can be obtained from the channel attribute lists by means of the `get_attrs` primitive), an appropriate prioritized scheme can be established in order to obtain the tuples from them.

3) Interaction Components: When programming a `SensorAgent`, besides the previously introduced TCS and TC primitives to communicate and synchronize with the actor, the programmer also needs a way of obtaining the sensor measurements. This is carried out by means of the following primitives:

- `read(type,cond,time_out)`. It returns a tuple with the sensor measurement obtained by the real sensor. The first argument indicates the type of sensor to be queried (`ACCELX, ACCELY, LIGHT, TEMPERATURE,...`). The second one established a condition that must be met (e.g. required value ranges). A time-out may be established to avoid an endless wait.
- `read_all(type,cond,time_out)`. It acts in the same way but it is used for a sensor acting as the root of a multi-hop sub-network in order to get not only the sensor measurement obtained, but also the different measurements of the rest of the nodes in the sub-network.
- `read_p(type,cond,attributes,tc_id,period, duration,time_out)`. It is used to achieve a periodic sending of sensor measurements through the specified channel (`tc_id`). Each tuple sent through the channel will have associated the specified `attributes`.
- `read_all_p(type,cond,attributes,tc_id,period, duration,time_out)`. It acts in the same way but it is used for a sensor acting as the root of a multi-hop sub-network.

The middleware architecture components that allow the realization of these operations are the `SensorDetectionAgent` and the `SensorInteractionAgent`. The former is in charge of detecting the different sensors deployed in a cluster and the latter acts as a relay station between the sensor agent and the real sensor. After detecting a sensor, the `SensorDetectionAgent` obtains its identifier and its geographic location, which is assumed to be known. Sensors can get their locations through GPS, or any number of other localization schemes, which is independent of the model and middleware proposed. If the application uses logical locations to deal with the sensors and actors instead of their identifiers, a *location file* must be created containing the relationship among the possible geographic locations taken into account for the application where the sensors and the actors can be deployed and their logical locations managed by the application code (for example, if a cluster is split in four quadrants logically known as N-E, N-W, S-E and S-W, the file should contain the real coordinates corresponding to each quadrant). The `SensorDetectionAgent` uses the obtained geographic location to access this location file and get the corresponding logical location. Then, it creates a pair `SensorAgent`/`SensorInteractionAgent` inside the actor host.

In order to deal with actor-actor interaction (following the hierarchical operational setting established), the middleware introduces two components: the `ActorAgent` and the `TCSProxy`. Let's consider an actor host within a cluster `A` acting as the leader actor of another actor host within a cluster `B` (see Fig. 2). These actors (i.e. the corresponding `ActorProcesses`) want to interact with each other. An `ActorAgent` deployed in the actor host in cluster `A` acts on behalf of the `ActorProcess` from the actor host in cluster `B`. The interaction between the `ActorAgent` of the host in cluster `A` and the `ActorProcess` from the host in cluster `B` is carried out through a `TCSproxy`. From the point of view of the `ActorProcess` of the host in cluster `B`, this proxy is accessed in the same way as a real TCS and it represents the TCS of the host in cluster `A`, which is the medium for communication with the `ActorProcess` of the host in cluster `A`.

The high-level coordination model supported is independent of the way the underlining infrastructure creates the clusters. So, clustering algorithms can range from topology-dependent to event-driven [15]. In the former, clusters are predetermined, depend on the network topology, and may be adaptively reconfigured to deal with mobility or failure of nodes. In the latter, cluster formation is triggered by an event so that clusters are created on-the-fly to optimally react to the event. The `ClusteringAgent` component appearing in the proposed middleware architecture is in charge of clustering issues and the creation of the corresponding `ActorAgents`. For the current prototype we have established a network topology so that cluster formation is predetermined.

B. On-Sensor Components

A `Connection` component pro-actively manages the connection phase. A sensor tries to establish a connection with an actor host by periodically sending a connection message. When the `SensorDetectionAgent` component detects one of these messages, it sends a confirmation message to the sensor, which will also send it back to the `SensorDetectionAgent` in order to finish the connection phase.

The remaining components form a reactive system, responding to incoming messages. A `Filtering` component receives all messages, eliminating incorrect messages and redirecting correct ones to the appropriate component: `Connection` or `RequestProcessing`. The latter extracts message parameters and determines whether the incoming request is either a recording request or a sensor query.

In the first case, the `RequestProcessing` component communicates with a `Logging` component. This component is only deployed in a sensor acting as the "root" of a multi-hop sub-network. The component is in charge of recording and updating sensed values coming from every member of the sub-network in order to process a further sensor query from the `SensorInteractionAgent` component when it is processing a collective (`read_all` or `read_all_p`) primitive calling. As future work, we are interested in improving the `Logging` component so that aggregation capabilities can be taken into account inside a sensor.

In the second case, the `RequestProcessing` component communicates with the corresponding sensor type (e.g. light) in order to get the sensed value, which is checked against the condition contained in the packet (if there is one). If the condition is met, a `QueryAnswering` component is required to assemble a packet containing the sensed value and pass

it on to the `GenericCommunication` component (a component provided by the operating system used), to be sent back to the actor host. In the case of a collective sensor query, the logged sensed values are also taken into account, as stated before, to form the answer.

IV. Implementation Details and Preliminary Results

In order to evaluate the middleware proposed, we are currently developing a prototype where laptops are playing the role of actors and Crossbow family motes [7] are being used as sensors. A mote consists of a MICAz processor and radio platform together with a MTS sensor board, both from Crossbow technologies. MICAz motes run TinyOS [20], a simple but highly concurrent open source operating system, which has been implemented by means of nesC [12], a C-based programming language for networked embedded systems, such as sensor networks. The necessary components we have deployed in the motes, previously described in section III-B, have been programmed by using nesC.

A Crossbow MIB510 serial interface board is connected to each laptop in order to communicate the actor with the motes. Java is used both as the programming language used to implement the middleware architecture components inside the different actor hosts (see section III-A) and as the base language used by the application programmer to develop the applications (that is, the language where TC-WSANs is integrated).

We have carried out some preliminary experiments in order to get a feel for the response time of the motes and the middleware overhead. The response times of the motes were analyzed by measuring the elapsed time between the sending of a request made by the actor through an one-to-many tuple channel and the reception of its first answer through a many-to-one channel. We carried out 1000 requests. For 1 mote the average of the response time was 26 ms. For 7 motes, this time was 29 ms. In both cases, most of the response times (94% and 84%, respectively), were less than 30 ms. A small amount of these times (3% for 1 mote and 3.5% for 7 motes) are on account of the middleware. Middleware overhead was even smaller (less than 2.5%) in another experiment where the motes periodically sent sensed data to the actor through a many-to-one channel. This minimal penalty overhead imposed by the middleware is acceptable because it is clearly compensated by the benefits of using a high-level coordination model for programming the interaction among the nodes of a WSANs.

On the other hand, the priority issue associated to the tuples within a channel has also been evaluated in order to demonstrate its suitability to tackle the real-time requirements of WSANs. In this experiment, a non prioritized event (temperature) was required every second and a prioritized event (noise level) was required every 5 seconds (a total of 70 prioritized events were considered). A time of 500 ms was taken into account to process a message on the actor side. This situation simulates a typical scenario where the system carries out a monitoring operation (e.g. temperature inside a building) and an important event is detected sporadically (e.g. a high noise level). Figure 3 compares the response time on the actor side to attend to the different prioritized events considering 2 and 7 motes and using both TC-WSANs and TCMote [9], a previous coordination model proposed for WSNs which does not take into consideration priority issues. These results show that in TC-WSANs, prioritized events are attended to quickly and that the instant in which the events occur or the number of motes do not matter. These are important factors contributing to the scalability of the system.

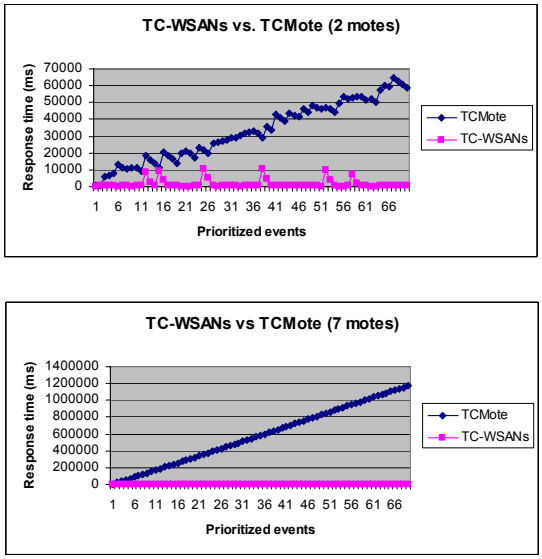

Fig. 3. Results to evaluate priority issues.

V. Conclusions

We have presented a middleware architecture for Wireless Sensor and Actor Networks. The reference operational setting, based on a (hierarchical) architecture of sensor clusters, each one governed by an actor has been described. The high-level coordination model supported by the middleware proposes communication and synchronization mechanisms based on tuple channels, which are priority queue structures that allow one-to-many and many-to-one communication schemes. The attribute-based naming scheme used together with the channel consumption behaviour enable the data-centric characteristics of typical sensor queries and satisfy the real-time requirements of WSANs. In addition, channel interconnection provides great flexibility for the definition of different interaction protocols. Some implementation details and preliminary results of the middleware prototype developed have been sketched.

References

[1] I.F. Akyildiz, I.H. Kasimoglu. “Wireless Sensor and Actor Networks: Research Challenges”. *Ad Hoc Networks Journal*, 2(4):351–367, 2004.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. “Wireless Sensor Networks: A Survey”. *Computer Networks Journal*, 38(4):393–422, 2002.

[3] J. Barbarán, M. Díaz, I. Esteve, D. Garrido, L. Llopis, B. Rubio, J.M. Troya. “TC-WSANs: A Tuple Channel based Coordination Model for Wireless Sensor and Actor Networks”. To appear in *Proceedings of the IEEE International Symposium on Computers and Communications (ISCC’07)*, Aveiro, Portugal, July 2007.

[4] N. Carriero, D. Gelernter. “Coordination Languages and their Significance”. *Communications of the ACM*, 35(2):97–107, 1992.
[5] P. Ciciriello, L. Mottola, G.P. Picco. “Building Virtual Sensors and Actuators over Logical Neighborhoods”. In *Proceedings of the 1st International Workshop on Middleware for Sensor Networks (MidSens’06), co-located with the 7th International Middleware Conference (Middleware’06)*, Melbourne, Australia, November, 2006.
[6] P. Costa, L. Mottola, A.L. Murphy, G.P. Picco. “TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks”. In *Proceedings of the 1st International Workshop on Middleware for Sensor Networks (MidSens’06), co-located with the 7th International Middleware Conference (Middleware’06)*, Melbourne, Australia, November, 2006.
[7] Crossbow Technology Inc. http://www.xbow.com/
[8] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, G.P. Picco. “TinyLime: Bridging Mobile and Sensor Networks through Middleware”. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communication (PerCom’05)*, pages 61–72. Kauai Island, Hawaii. IEEE Computer Society Press, 2005.
[9] M. Díaz, B. Rubio, J.M. Troya. “TCMote: A Tuple Channel Coordination Model for Wireless Sensor Networks”. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS’05)*, pages 437–440. Santorini, Greece. IEEE Computer Society Press, 2005.
[10] D. Estrin, R. Govindan, J. Heidemann, S. Kumar. “Next Century Challenges: Scalable Coordination in Sensor Networks”. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom’99)*, pages 263–270, Seattle, WA, USA, 1999.
[11] C-L. Fok, G-C. Roman, C. Lu. Rapid Development and Flexible Deployment of Adaptative Wireless Sensor Network Applications”. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS’05)*, pages 653–662, Columbus, Ohio, USA. IEEE Computer Society Press, June 2005.
[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler. “The nesC Language: A Holistic Approach to Networked Embedded Systems”. In *Proceedings of Programming Language Design and Implementation (PLDI’03)*, 2003.
[13] J. Gehrke, L. Liu (Guest Eds). “Sensor-Networks Applications”. *IEEE Internet Computing*, 10(2), 2006.
[14] D. Gelernter. “Generative Communication in Linda”. *ACM Transactions on Programming Languages and Systems*, 7(1), 80–112, 1985.
[15] T. Melodia, D. Pompili, V.C. Gungor, I.F. Akyildiz. “A Distributed Coordination Framework for Wireless Sensor and Actor Networks”. In *Proceedings of ACM Mobihoc 2005*, Urbana-Champaign, ll, USA, 2005.
[16] L. Mottola, G.P. Picco. “Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks”. In *Proceedings of the 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS’06)*, San Francisco, CA, USA, 2006.
[17] C-C. Shen, C. Srisathapornphat, C. Jaikaeo. “Sensor Information Networking Architecture and Applications”. *IEEE Personal Communications*:52–59, 2001.
[18] F. Silva, J. Heidemann, R. Govindan, D. Estrin. “Directed Diffusion”. In Iyengar, S. and Brooks R. R. (eds), *Frontiers in Distributed Sensor Networks*, CRC Press, 2004.
[19] SMEPP. http://www.smepp.net/
[20] TinyOS. http://www.tinyos.net/