TC-WSANs: A Tuple Channel based Coordination Model for Wireless Sensor and Actor Networks [*]

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. 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. This paper introduces TC-WSANs, a high-level coordination model that addresses these two requirements and facilitates the application programmer task. Our proposal is based on a (hierarchical) architecture of sensor/actor clusters and the use of tuple channels to achieve communication and synchronization among sensors and actors. 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 characteristics of TC-WSANs and the primitives that it provides for its integration into a computational host language are presented.

1. Introduction

Wireless Sensor and Actor Networks (WSANs) is rapidly attracting interest among researchers and practitioners [1]. They refer to a group of sensors and actors linked by a wireless medium to carry out phenomenon sensing and acting tasks. The devices deployed in the environment are, on the one hand, tiny, low-cost and low-power sensor nodes able to sense environmental data, and, on the other hand, resource rich 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. 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. WSANs have been successfully used for various application areas, such as environmental monitoring, object and event detection, military surveillance, precision agriculture [9].

WSANs have two major requirements:

- *Coordination*. Coordination mechanisms are needed for both sensor-actor and actor-actor interactions. 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.
- *Real-time*. 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 communication is very important in WSANs. Thus, coordination models and communication protocols should support real-time properties for this kind of system.

This paper presents TC-WSANs, a high-level coordination model to facilitate application development over WSANs. The model is based on TCMote [6], a previous coordination model proposed for *Wireless Sensor Networks* (WSNs), i.e. networks where only sensor nodes are deployed [2]. TC-WSANs adapts TCMote and extends it with real-time characteristics in order to satisfy the above mentioned requirements in the new target systems. Our reference operational setting is based on an architecture where

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

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.

Both sensor-actor and actor-actor coordination are based on the use of *Tuple Channels* (TCs). 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. 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.

Much work has targeted the development of coordination models and supporting middleware in the effort to meet the challenges of WSNs [5] [7] [11]. However, since the above listed requirements impose stricter constraints, they may not be suited to be applied to WSANs. To the best of our knowledge, less work proposes coordination models providing high-level constructs to ease the application programmer task and to afford the unique features and requirements of WSANs. A recent related work, called TeenyLime, can be found in [3]. TeenyLime is based on shared tuple spaces [10].

The rest of the paper is structured as follows. In Section 2 the reference operational setting is described. Section 3 presents the TC-WSANs model, including its main design goals, concepts and primitives. In Section 4 some implementation details and preliminary results of the current prototype we are developing are given. Finally, some conclusions and future work are sketched in Section 5.

2. Operational Setting

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, 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,

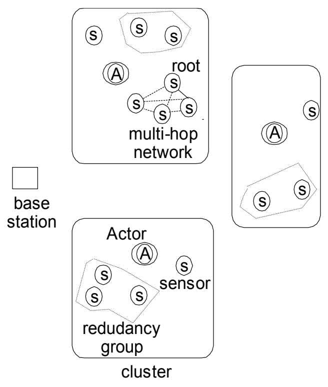

Figure 1. Operational setting.

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 (Fig. 2). 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.

3. 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 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 our model, the entities to be coordinated E are the nodes, i.e. actors and sensors. Each actor owns a Tuple

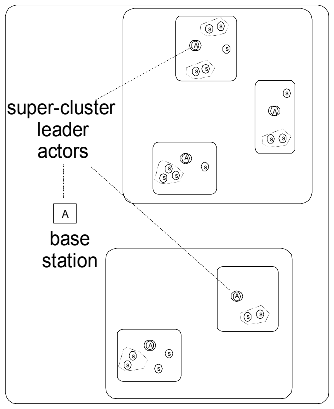

Figure 2. Hierarchical structure.

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 (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, ...).

In the following sections, we are going to describe, in a detailed way, the characteristics of TC-WSANs and the primitives that it provides for its integration into a host language in order to support sensor and actor network applications.

3.1. 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.

To facilitate the data-centric characteristics of sensor queries, attribute-based naming is the scheme selected [11] [12]:

```
[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]
```

Besides the attributes specified 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 (see section 3.3).

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.

3.2. Tuple Channel

As stated before, 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. In Fig. 3, thick lines represent channels and thin lines represent channel accesses. 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 router mechanism is independent of the TC-WSANs model. Only the root sensor of this network will use channels to get in contact with the cluster actor 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 or a value of any established data type of the host language where the model is integrated.

Entities access a 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.

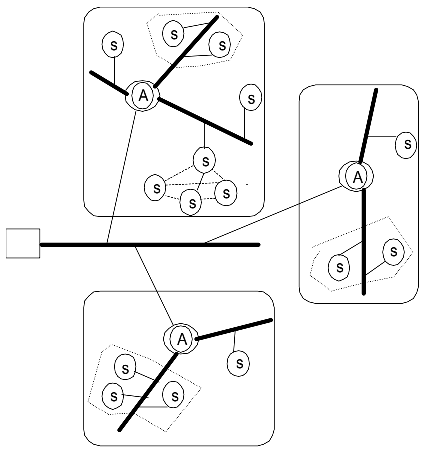

Figure 3. Communication through channels.

- 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, 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.

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. 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_attrs primitive. This way, the entity can establish a priority-based scheduling to get data from different channels.

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 spec-

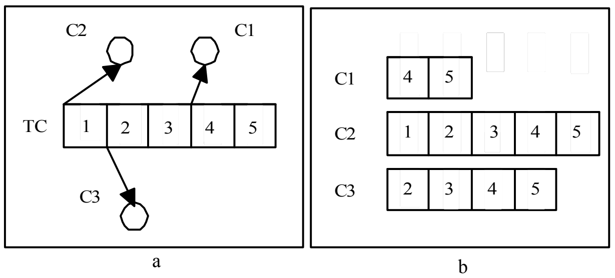

Figure 4. Consumption behaviour.

ified 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.

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. 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 channel TC.

Both together 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 behaviour

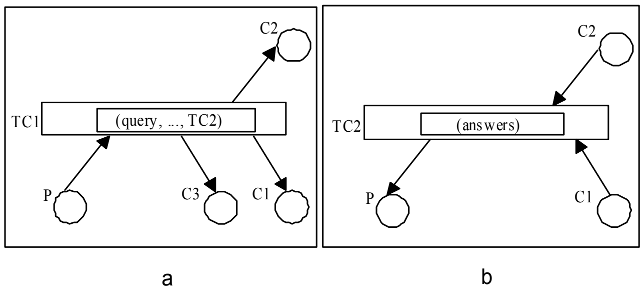

Figure 5. One-to-many and many-to-one communication schemes.

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.

3.3. 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 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, the appropriate prioritized scheme can be established in order to obtain the tuples from them.

4. Implementation Details and Preliminary Results

In order to evaluate the high-level coordination model proposed, we are currently developing a prototype where laptops are playing the role of actors and Crossbow family motes [4] are being used as sensors. The necessary components we have deployed in the motes have been programmed by using nesC [8]. Java is used both as the programming language used to implement the system components inside the different actor hosts 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 WSAN.

5. Conclusions and Future Work

We have presented TC-WSANs, a coordination model for Wireless Sensor and Actor Networks. The model communication and synchronization mechanisms are based on tuple channels, which are priority queue structures that allow one-to-many and many-to-one communication schemes that together with the used attribute-based naming scheme 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.

References

[1] I. F. Akyildiz and 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, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks Journal*, 38(4):393–422, 2002.

[3] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks. In *Proceedings of the 1st International Workshop on Middleware for Wireless Sensor Networks (MidSens 2006), co-located with the 7th International Middleware Conference (Middleware'06)*, pages 43–48, Melbourne, Australia, November 2006. ACM Press.

[4] Crossbow. http://www.xbow.com/.

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

[6] M. Díaz, B. Rubio, and J. M. Troya. TCMote: A Tuple Channel Coordination Model for Wireless Sensor Networks. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS 2005)*, pages 437–440, Santorini, Greece, July 2005. IEEE Computer Society Press.

[7] C.-L. Fok, G.-C. Roman, and 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 2005)*, pages 653–662, Columbus, Ohio, USA, June 2005. IEEE Computer Society Press.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI 2003)*, pages 1–11, San Diego, California, USA, June 2003. ACM Press.

[9] J. Gehrke and L. L. (eds). Sensor-network applications. *IEEE Internet Computing*, 10(2), 2006.

[10] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *IEEE Personal Communications*, pages 52–59, August 2001.

[12] F. Silva, J. Heidemann, R. Govindan, and D. Estrin. *Frontiers in Distributed Sensor Networks*, chapter Directed Diffusion. CRC Press, Inc., Boca Raton, Florida, USA, October 2003.