TCMote: A Tuple Channel Coordination Model for Wireless Sensor Networks

Manuel Díaz, Bartolomé Rubio and José M. Troya
Dpto. Lenguajes y Ciencias de la Computación.
Málaga University
29071 Málaga, SPAIN
{mdr,tolo,troya}@lcc.uma.es

Abstract

Wireless sensor networks are potentially one of the most important technologies of this century. Sensor nodes must carry out a coordinated cooperative effort in order to transmit the required data. This paper introduces TCMote, a new coordination model to facilitate application development over ad hoc sensor networks. Our proposal is based on a (hierarchical) architecture of sensor regions and the use of tuple channels to model communication and synchronization among sensors. A tuple channel is a FIFO structure that allows data structures to be communicated both in an one-to-many and many-to-one way, facilitating the data-centric behavior of sensor queries. The characteristics of TCMote and the primitives that it offers for its integration into a computational host language are presented.

keywords. *Wireless Sensor Networks, Coordination Paradigm, Coordination Models and Languages, Coordination Middleware.*

1. Introduction

Wireless sensor networks constitute a new pervasive technology. Due to a combination of advances in electronics, nanotechnology, wireless communications, computing, networking, and robotics, it is now possible to design advanced sensors and sensor systems that can be used for various application areas: environmental monitoring; objects and events detection; military surveillance; precision agriculture; transportation [1].

A sensor network consists of (potentially) thousands of tiny, low-cost and low-power nodes, colloquially referred to as "motes", that interact with their environment through sensors, actuators and communication. Sensor nodes must carry out a coordinated cooperative effort in order to transmit the required and partially processed data [7].

These coordination needs has attracted the coordination paradigm community attention. Coordination models and languages facilitate application development by providing high-level constructs such as tuple spaces, blackboards, and channels, and they have been successfully applied to a wide range of application areas: open systems [11], interactive web environments [2], scientific computation [6], just to mention some of them. Recently, different approaches are coming to the sensor network area [8] [4], most of them based on principles of Linda [10], which is historically the first genuine member of the family of coordination languages and is based on a shared multiset of tuples referred to as tuple space.

This paper presents TCMote, a new high-level coordination model based on *tuple channels* to facilitate application development over wireless sensor networks. Our reference operational setting is first described. It is based on a (hierarchical) architecture of sensor regions, each one governed by a leader with higher capabilities (power, memory, processing ability) than the rest of the region nodes (motes). The leader manages both internal and external communications. Besides the deployment of new motes, these can move inside a region or even shift region, reconfiguring the system in order to get a better performance.

A region leader owns a *tuple channel space*, which stores tuple channels used to carried out communication and synchronization between region sensors and the leader in a single-hop way. Tuple channels are based on and simplify those introduced in TCM [5], a previous coordination model for parallel and distributed programming. A TCMote tuple channel is a FIFO structure that allows one-to-many and many-to-one communication of data structures, represented by tuples.

The rest of the paper is structured as follows. In Section 2 the reference operational setting is described. Section 3 presents the TCMote model, including its main design goals, concepts and primitives. Finally, some conclusions and future work are sketched in Section 4.

2. Operational Setting

Most sensor network middleware operate in a setting where the sensors are fixed in the environment and report their values to a centralized point, requiring multi-hop communication among sensors. However, this fixed and centralized model may not be the ideal for all applications. Consider a city pollution measurement application. People from different city quarters can be provided by a sensor. A person may be required to move inside the quarter or even shift quarter, in order to obtain more precise measurements. In order to get energy-efficiency and scalability, instead of report the information to the central base station in a multi-hop way, sensors communicate directly to a quarter base station, which will be in charge of establishing communication with other quarter base stations and with the central base station.

We propose an operational scenario that fits this kind of applications where motion and single-hop communication among nodes are required. It is based on an architecture of sensor regions, each one governed by a leader with higher capabilities (power, memory, processing ability) than the rest of the region nodes (motes). Motes can move inside a region or even shift region, and new ones can be deployed, reconfiguring the system in order to get a better performance.

Due to the sheer numbers of nodes involved in a sensor network, some degree of redundancy can be expected, improving reliability and easing the self-configuration process. In our operational setting, several motes may form a redundancy group inside a region. The zone of the region monitored by a redundancy group is covered in spite of failures or sleeping periods of the group members.

A hierarchical structure may be achieved clustering different regions into a super-region, whose member nodes are the corresponding region leaders, one of which will also act as the leader of the super-region. In our operational setting, the base station can be considered as the leader of a region grouping leaders of outermost regions.

3. The TCMote 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 constrains, ...).

In our model, the entities to be coordinated E are the nodes (leaders and motes) of a sensor network fitting the operational scenario described in the previous section. A region leader owns a tuple channel space, which constitutes the coordination media M. It stores tuple channels used to carry out communication and synchronization between region sensors and the leader. 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, ...).

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

3.1. Tuple Channel Spaces

A tuple channel space is a shared data space accessed by the members of a region. Every region has one. As stated before, the region leader is the one that owns it. The tuple channel space interface offers the following primitives in order to interact with it:

- `store(attributes)`. It stores a channel in the tuple channel space. It is a non-blocking primitive.
- `withdraw(search_pattern,time_out)`. It withdraws one or several channels from the tuple channel space. It is a blocking primitive. A time-out can be established to avoid an endless wait.
- `find(search_pattern,time_out)`. It finds a channel in the tuple channel space and obtains information about it. It is a blocking primitive. A time-out can also be established.
- `react(operation,reaction)`. It associates a reaction to a tuple channel space operation. It is a non-blocking primitive.

When a tuple channel is stored in the tuple channel space, not only its identifier is specified, but also its attributes, i.e. some information that can be useful in order to establish some communication. Actually, the identifier is itself a channel attribute. In general, inside the tuple channel space, a tuple channel is specified by the following attribute-based data structure:

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

Both `withdraw` and `find` primitives use as first argument a `search_pattern`, i.e. an attribute-based data structure with (probably) some partially specified fields. A pattern matching process is carried out in order to find channels with attributes matching the search pattern. In the case of the withdraw operation, all found channels are withdrawn from the tuple channel space. For the find operation, if several candidates are found, one of them is chosen in a non-deterministic way. The partially specified fields are instanti-

ated with the corresponding values from the found channel information.

A reaction can be associated to a tuple channel space operation by means of the `react` primitive, which has two arguments: the implied operation and the reaction it is desired to associate to 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 tuple channel space, a failed reaction yields no result at all. The operations we have initially considered as candidates to be included inside a reaction are:

- `store_r(attributes)`.
- `withdraw_r(search_pattern)`.
- `find_r(search_pattern)`.
- `no_r(search_pattern)`.

The three first primitives have the same effect as the previously specified ones, but they are non-blocking primitives. So, `withdraw_r` and `find_r` succeed when the search pattern matches any data structure in the tuple channel space, but fail otherwise. Complementary to `find_r` and `withdraw_r`, `no_r` succeeds when its argument does not match any data structure in the space, but fails otherwise.

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

3.2. Tuple Channels

A tuple channel is a FIFO structure that allows one-to-many (from the region leader to some region nodes) and many-to-one (from some region nodes to the region leader) communication.

Data structures communicated through channels are represented by tuples. A tuple has the form:

$$(t_1, t_2, ..., t_n)$$

where t_i can be:

- a tuple channel identifier.
- a value of any established data type of the host language.
- a tuple holder (see Section 3.3).

Nodes access tuple channels by means of four primitives:

- `connect(tc_id)`. The node that executes it establishes a connection to the specified channel. It is a non-blocking primitive.
- `disconnect(tc_id)`. The specified channel is disconnected from the node executing the primitive. It is a non-blocking primitive.
- `put(tc_id,tuple)`. It sends a tuple through a channel. It is a non-blocking primitive.
- `get(tc_id,time_out)`. It obtains a tuple from a channel. It is a blocking primitive. A time-out can be established to avoid an endless wait.

Before a node can send/receive information through/from a tuple channel, it must establish a connection by means of the `connect` primitive. On the other hand, when a node does not need a channel any more, it executes the `disconnect` primitive in order to disabling the tuple channel connection. Connection information will be useful to the run-time system in order to achieve an efficient channel implementation.

A producer node 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 end of the channel. It is an asynchronous primitive. In our model, a channel has an unlimited size.

A consumer node 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. It is a blocking primitive, i.e. the node executing it will suspend if the channel is empty. The `get` primitive does not withdraw any data from the channel from the point of view of the other consumers and so, they can also consume the same information. We can say that each consumer node accessing a channel will have, at any time, a view of it, which may be different from the views of the rest of consumers sharing the channel.

Many-to-one communication scheme is useful in order to continuously send values from sensors to a region leader. On the other hand, both together one-to-many and many-to-one communication schemes are appropriate to deal with typical queries in sensor networks such as “which region/area/quadrant has temperature higher than $40°C$?”, “what is the average temperature in each region?”. The `get` primitive behavior allows that every region/area/quadrant can receive the same query (one-to-many). Implied nodes can answer it (many-to-one) by means of (and after connecting to) a new channel whose identifier was sent inside the tuple representing the query.

In addition to information communication, channels can also be used to control communication. For example, messages such as “move to northeast quadrant”, “change to region i” can be sent to a mote to reconfigure the system in order to get a better performance.

3.3. Tuple Holders

Besides the communication and synchronization scheme through tuple channels described above, TCMote also provides other useful mechanism for information exchange based on the use of *tuple holders*. A tuple holder is a special kind of single-assignment variable accessed by means of two primitives:

- `write(th_id,tuple)`. If the tuple holder `th_id` is unbound, it is instantiated to `tuple`; nothing is done otherwise. It is a non-blocking primitive.
- `read(th_id,time_out)`. If the tuple holder `th_id` is bound, it reads the corresponding tuple; it blocks otherwise. The same tuple holder can be read by different consumers. A time-out can be established to avoid an endless wait.

Unbound tuple holders can be sent through a channel (as a tuple field) to achieve backward communication. This allows an elegant way of implicit and direct communication. The tuple consumer is the one that will instantiate the tuple holder in order to answer some request, avoiding the use of another channel.

This mechanism can be useful to deal with redundancy groups, where the answer of any mote may be enough to obtain the required information about the zone controlled by the group. Moreover, a region leader could answer to base station queries such as "what is the average temperature in the southeast quadrant of region i?" through a tuple holder. In addition, the use of tuple holders supports an event handling mechanism, which is suitable for sensor network applications where sensor nodes are programmed to process asynchronous events such as receiving a message or an event triggered by a timer.

4. Conclusions and Future Work

We have presented TCMote, a Tuple Channel-based coordination Model. It provides the means for developing wireless sensor network applications. The reference operational setting, based on a (hierarchical) architecture of sensor regions, each one governed by a leader with higher capabilities than the rest of the region nodes, has been described. The model communication and synchronization mechanisms are based on tuple channels, which are FIFO structures that allow data-centric sensor queries to be carried out in an elegant way.

We are currently developing a middleware prototype for supporting TCMote. We are using Java as both the host language (where the tuple channel model is integrated) used by the application programmer and the programming language used to implement the middleware architecture components inside the different region leader hosts. On the other hand, nesC [9] is being used to program the necessary components inside the motes. Crossbow Micaz motes [3] are being used. Two different collaborative enterprises have shown interest in developing two initial applications in order to carry out radiation measurements in stopping periods in a nuclear power plant and to analyze the environmental pollution in a city.

References

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

[2] G. Cabri, L. Leonardi, F. Zambonelli, "MARS: a Programmable Coordination Architecture for Mobile Agents", *IEEE Internet Computing*, 4(4), 2000, pp. 26–35.

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

[4] 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 2005)*, IEEE Computer Society, Kauai Island, Hawaii, 2005.

[5] M. Díaz, B. Rubio, J.M. Troya, "The Tuple Channel Coordination Model", in *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed systems (PDSE 1997)*, IEEE Computer Society, Boston, USA, 1997, pp. 95–106.

[6] M. Díaz, B. Rubio, E. Soler, J.M. Troya, "A Border-based Coordination Language for Integrating Task and Data Parallelism", *Journal of Parallel and Distributed Computing*, 62, 2002, pp. 715–740.

[7] 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 1999)*, Seattle, WA, USA, 1999, pp. 263–270.

[8] C-L. Fok, G-C. Rooman, G. Hackmann, "A Lightweight Coordination Middleware for Mobile Computing", in *Proceedings of the 6th International Conference on Coordination Models and Languages (COORDINATION 2004)*, LNCS vol. 2949, Pisa, Italy, 2004, pp. 135–151.

[9] 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 2003)*, 2003.

[10] D. Gelernter, "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems*, 7(1), 1985, pp. 80–112.

[11] R. Tolksdorf, "Laura: A Service-based Coordination Language", *Science of Computer Programming*, 31(2-3), 1998, pp. 359–381.