Proceedings of the

Sixth Annual IEEE International Conference on Pervasive Computing and Communications

17-21 March 2008, Hong Kong

CPS Conference Publishing Services

Los Alamitos, California

Washington • Tokyo

IEEE computer society

Sixth Annual IEEE International Conference on Pervasive Computing and Communications

PERLA: a Data Language for Pervasive Systems

F.A. Schreiber, R. Camplani, M. Fortunato, M. Marelli, F. Pacifici
Politecnico di Milano, Dipartimento di Elettronica e Informazione, Milano, Italy
{schreibe, camplani}@elet.polimi.it, filippo.pacifici@polimi.it

***Abstract*—A language is presented for managing data in highly pervasive systems made of very different devices as to their technology and functional capabilities. Functional and non-functional requirements are dealt with in a transparent mode by a SQL like interface. In this paper the most relevant features of the language, the related data structures and some query examples are briefly introduced.**

I. INTRODUCTION

In a very short time, the interest for Wireless Sensors Networks and their applications has grown both within the academic community and, even if with some warnings [1], in the world of actual users. At the same time, the complexity of the envisaged WSN-based systems grew from a handful of homogeneous sensors to hundreds or thousands of devices differing as to their capabilities, architectures, and languages.

The result of this inflationary expansion is the difficulty an application programmer encounters in dealing with the languages and protocols, which characterize different portions of the system, and in optimizing sampling, storage, and transmission strategies in order to save as much as possible of power in the batteries, which are the most wearable components in a system of mostly unattended small devices.

As a motivating example of such systems, we refer to one of the case studies in the ART DECO project [2], a large project funded by the Italian University Ministry: the automation of a large wine production farm *from the vineyard to the table*. Table I schematically shows information items and devices used for supporting the different phases of the production and delivery processes.

TABLE I
DEVICES USAGE AND TYPES

WHERE	WHAT	HOW
vineyard	humidity, temperature, chemicals	sensors
cellar	humidity, temperature	sensors
bottle	tracking information	RFID tag
pallet	tracking information, temperature	RFID tag, sensors
truck	position information	GPS
workers	information system	PDA

A number of efforts have been made to define and implement a High Level Language for managing data in WSN applications, some of the most noticeable among them are TinyDB, GSN, and DSN.

TinyDB [3] [4] is one of the first efforts in querying WSN data with a SQL like interface; multiple persistent queries with different sampling time are issued from a pc, data are collected from Motes sensors in the environment, filtered, possibly aggregated, and routed out to a base station. It is defined over TinyOS and it exploits power-efficient in-network processing algorithms. Its portability is bound to that of TinyOS.

GSN [5] [6] is a scalable, lightweight system which can be easily adapted, even at run-time, to new types of sensors, thus allowing a dynamic reconfiguration of the system. XML is used as the network and data specification language, while SQL is used as the data manipulation language.

DSN [7] [8] uses a completely different approach. The whole system is built in the declarative language Snlog - a dialect of Datalog - both for data acquisition and for network and transmission management.

The large heterogeneity of the pervasive systems we are dealing with, comprising devices ranging from passive RFID tags to application servers, requires a very high level of transparency so that as much as possible of the technical problems be handled by the system, while the application programmer only writes high level code. This observation led us to the approach *"single system - single language"*, but, unlike in DSN, we stay as near as possible to SQL, since it is still the most widely known and used data language.

At the outset, we only thought to implement a multi-platform version of TinyDB, which could be deployed on different sensor families with little effort; however, during the analysis phase, we extended our goal in three successive steps:

- *run time* support of heterogeneity;
- support of *non intelligent* device classes, such as RFID tags;
- extend the target to *generic pervasive systems*.

While the impact of the first step is limited to the middleware, which must provide a set of APIs to manage the different platforms in a uniform way, the other two steps require a rethinking of the functional features of the language. Tags are not equipped with sensors and cannot perform data manipulation or transmission, but their interaction with the external world is mediated by the RFID reader while the exchanged information is the tag presence itself, and not some sensed value. Therefore, we must provide an abstraction for the RFID behavior and this can be done in two ways: the first considers an *RFID tag as it were a sensor* whose "sampled data" is the *identifier of the last reader* which sensed the tag. The second one considers the *RFID reader as it were a sensor* whose "sampled data" is the *identifier of the last tag* sensed by the reader. Since communications with the network are always handled by the reader, the first solution provides a higher level of abstraction and requires a more sophisticated middleware. Moreover, the presence of RFIDs claims for the introduction of an *event based semantics* at the language level, since we cannot rely on a fixed sampling interval, but actually "sampling" is performed by the reader at the moment of the tag passage.

The third step has been made possible thanks to the availability of a logical and distribution independent software platform for large heterogeneous networks, described in section 2, provided by another workgroup of the ART DECO project [9]. This allowed our work to focus on the high level declarative language definition, being freed from low level programming issues, and processing the queries in terms of logical objects abstraction.

The language has been defined in order to manage both *functional features*, comprising the definition of operations which manipulate raw data to generate the query output and statements for the setting of sampling parameters, and *non-functional features*, which account for constraints on the offered functionalities and on the Quality of Service (QoS); in WSNs the QoS is mainly related to power management, however node latency and sensors availability are considered as well. Non-functional features are dealt with through a "policies" mechanism, which will be explained in section 3.

The heterogeneity of the considered devices classes led us to identify two levels of abstraction represented by two levels in the language:

- a *Low Level Language* whose goal is to manage the sampling operations performing only data manipulations on a single sensor;
- a *High Level Language* whose role is that of manipulating sampled data in order to produce queries results.

Both languages have a SQL-like syntax; however, the semantics of the Low Level Language differs, since it can be thought as a mechanism to generate the data streams as in figure 1, and to determine when sampling should be performed on which nodes.

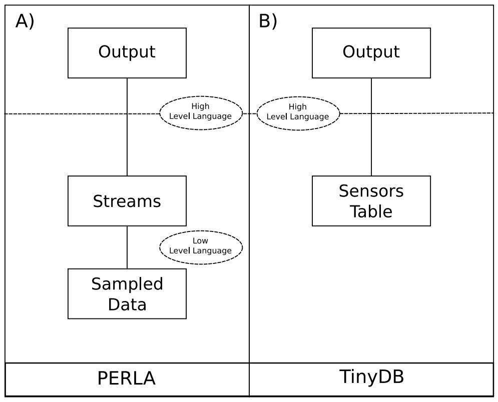

Fig. 1. Comparison between PERLA and TinyDB.

The rest of the paper is organized as follows: section 2 presents the architecture of the system and its middleware

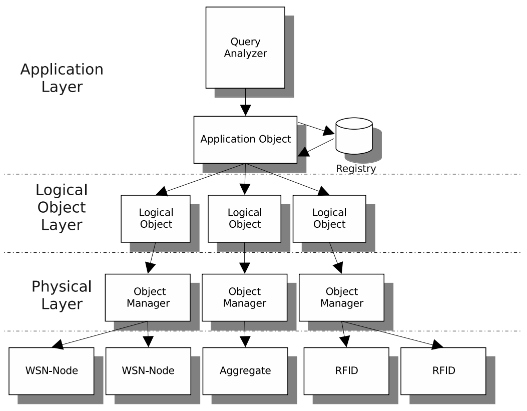

Fig. 2. Middleware architecture.

components; section 3 introduces the PERLA language functional and non-functional features; in section 4 the query processing mechanism is explained together with examples - drawn from the above mentioned case study - which show the specificities of our approach; finally, section 5 presents the current state of the project and the work still to be done.

II. System architecture and middleware

In this section the architecture for pervasive systems, over which we based our language, is briefly explained. The abstraction levels provided by the architecture and the interface supporting the two levels of the language are presented.

Figure 2 shows the system architecture:

- *Application Layer*: is the front-end used by applications in order to access data coming from the physical devices;
- *Logical Objects Layer*: provides an abstraction for physical devices;
- *Device Access Layer*: provides the underlying infrastructure for accessing devices, abstracting from the required software distribution.

In the application layer, the *query analyzer* handles user-submitted queries and, using a dedicated component (i.e., *the registry*), it retrieves information about logical objects composing the system. Thus, the query analyzer selects logical objects relying on this information to execute the query.

In the logical layer, each component (i.e., logical object) wraps single or homogeneous groups of devices. In the first case, a logical object hides the complexity for accessing devices (in this way, a change in lower layers is transparent to the higher ones). In the second case, besides hiding the physical devices composing the aggregate, the operation manager directly provides aggregated results to the logical object.

Logical objects expose a standard interface that unifies and simplifies the access to higher layers. In particular, they provide three categories of attributes:

- *Static attributes* represent constant values describing a characteristic of the node (i.e., node type, maximum sampling rate, etc.);

- *Dynamic probing attributes* are the variables that a logical object must read from physical devices (i.e., a sensor measurement);
- *Dynamic non probing attributes* are those for which a logical object can return a local cached value without actually dealing with the physical device (i.e., current base station).

Note that different logical objects can expose different sets of attributes and that the same attribute can be dynamic for some devices and static for other ones (e.g. location). Logical objects interface can also provide events used to signal changes in the physical devices. Non probing attributes often have an associated event that notifies a change of their values (e.g. last sensed RFID reader changed).

The implementation of logical objects will be deployed as much as possible on the lowest layers of the architecture: if a device is too tiny and has very poor computational resources, the logical object wrapping it will be deployed at a higher layer.

The device access layer provides the access point to devices allowing to abstract the software infrastructure required by a specific device technology. For instance, in RFID-based systems, the operation manager is the middleware used to drive the reader.

III. THE LANGUAGE FEATURES

We classified the features that should be exploited by the language in four categories:

- *Data representation*. The language should be able to hide physical devices as much as possible and to provide a database view of the whole pervasive system: statements written by users are queries on this database. Differently from traditional databases, every query must also specify how, when and where the sampling operations should be executed.
- *Physical devices management*. The main issues in providing physical devices abstraction are the definition of the sampling semantics for each class of devices and the introduction of a temporal semantics taking into account the existence of losses and delays in physical communications. With reference to the architecture presented in section 2, each device is abstracted by a logical object and each sampling operation on the device is abstracted by the reading of a logical object attribute. Two types of sampling are supported: periodic or event based; the latter is activated when a logical object event is fired (typically used in RFID devices).
- *Functional characteristics*. Language statements should allow the user to specify sampling parameters (time, mode, etc.) and the set of operations to manipulate raw data in order to generate a query output.
- *Non functional characteristics*. In WSNs the most important non functional parameters are related to power management. However, due to the node heterogeneity we are considering, a generic approach must be introduced: a small number of clauses should be provided by the language to support all the non functional characteristics that can be considered now or discovered in the future. From the user point of view there are no differences between logical objects attributes that retrieve data of interest and logical object attributes that return non functional parameters. However, an important difference exists from the system point of view: non functional fields exposed by logical objects are expressed in an abstract way. Then, they are internally translated into concrete values that can be handled by physical devices. For instance, a percentage power level attribute can be obtained from the voltage value for a certain class of devices, predicted from the number of executed operations for other devices, or just set to 100% for AC powered devices. Non functional attributes can be used to decide if a node should participate to a query, to set the current sampling rate, to retrieve information about network nodes, etc.

We now briefly outline the functionalities that should be supported by logical object interfaces of physical devices involved in query execution, so that the language semantics can be then explained in terms of the interaction between the query analyzer and the logical objects. The interface must provide three kinds of functionalities:

- *Retrieving attributes values:* attributes can be both data of interest and policy values. A special ID attribute must be supported by all the logical objects to allow the query analyzer to univocally identify them.
- *Firing notification events*: events can be used to perform event based sampling or to activate query selections.
- *Getting the list of supported attributes*: a set of methods should be provided to allow the query analyzer to know the set of attributes (and their data types) a logical object exposes.

In the following the main language issues are outlined.

A. Data structures

The data structures which support our language are the *stream tables* (or *streams*) and the *snapshot tables* (or *snapshots*). Streams are unbounded lists of records produced by queries. Each record has a set of user defined fields and a native timestamp field. It supports the *insertion* and the *reading* operations. In *insertion*, new records can be inserted into the stream by a running sub-query. The execution of this operation generates an *insertion event* that can be detected and used by other sub-queries. In *reading*, a data window can be extracted from the stream by a running query. This window is defined by a timestamp value and by a size (i.e., number of records).

The snapshot table is a set of records produced by a query in a given period. During this period, all the records generated are stored in a buffer. At the end of the period the snapshot table is filled with records from the buffer. When the snapshot is read, the current content is returned.

B. High and Low Level Queries

As pointed before, we defined two SQL like languages: the Low and High Level Language. A user submitted query is composed of some Low Level and some High Level Queries, each of them having the role of retrieving and manipulating data and inserting the produced results in a data structure. A query written with the Low Level Language is used to define the behavior of a single device (or a group of devices abstracted by a single logical object). The main role of Low Level statements is allowing a precise definition of the sampling operations, but also allowing the application of some SQL operators to sampled data. An object that is executing a Low Level Query should maintain a local buffer and perform the following activities:

- to sample data by reading some logical object attributes and inserting the read values into the local buffer.
- to perform SQL operations (selection, aggregation, filtering, grouping, etc.) on the current content of the local buffer and insert the obtained records into the output data structure.

The query can request the execution of both the previous activities, periodically or when an event happens. The local buffer is conceptually unbounded and its size increases indefinitely. Practically, the executor should be able to do a garbage collection of records which are no longer required by SQL operations. It is worth to note that all the processing executed at low level is relative to data extracted from a single logical object and has the goals of discarding bad values and optionally aggregating a group of sampled values before sending them to High Level Queries. If a node cannot process data or maintain a buffer (e.g. an RFID tag) the query will be executed on another physical device (e.g. the RFID reader), but this distribution is hidden to the language. A Low Level statement can contain also an `Execute If` clause allowing the definition of the set of logical objects that will execute the query, in terms of conditions on logical object attributes. High Level Queries take one or more *streams* (generated by Low Level Queries or other High Level Queries) as input, perform SQL operations on windows extracted from *input streams* and insert the generated records in an output data structure. The activation of a High Level Query can be specified either in terms of a time period or in terms of an event (insertion of a record into a stream). Note that the High Level Language has similar functions as the TinyDB language, because both are used to manipulate *data streams* coming from sensors. The Low Level Language has not a counterpart in TinyDB and can be thought as a mechanism to *generate data streams* corresponding to the TinyDB sensors table (see figure 1).

C. Pilot join operation

Analyzing some case studies, we realized that in many real situations *a sampling on a node should be started if and only if a certain value has been retrieved from a sampling done on another node*. For example, suppose that a user requires a temperature monitoring of all the wine pallets placed in the trucks that are currently in a given parking area. If temperature sensors are mounted on the pallets, the sampling operation on a node should be activated only when its truck is in the parking area. Truck locations are detected by position sensors, that are nodes physically different from the temperature nodes and not directly connected to them.

The above consideration suggested us that a specific operation should be supported to allow sampling activation depending on data sampled from other nodes. We called this new operation *Pilot Join*, because it is conceptually similar to the SQL join operation, but it is used to activate the execution of a Low Level Query on logical objects.

Two kind of Pilot Join are possible:

- *Event based Pilot Join*. When an event happens (i.e. a record is inserted into a stream) a given set of nodes should start sampling (typically for a fixed period). Suppose that, in the previous example, the temperature must be sensed once every time a truck enters the parking area; in this case an event based pilot join is required.
- *Condition based Pilot Join*. A continuous sampling (with frequency f) should be done on all the nodes that are connected to a base station that is in a list of base stations satisfying given criteria. This list is periodically updated with a frequency lower than f: this behavior is obtained using a snapshot data structure. Consider again the previous example. Suppose that a running query is sensing (with low frequency) the position of all the trucks and inserting them into a stream. Suppose also that the required behavior of the system is the continuous sampling of temperature sensors mounted on pallets whose last monitored position is in the parking area. In this case a condition based pilot join must be used.

IV. Query processing and examples

A user-defined query can be expressed as a graph: nodes are either data structures or queries, while edges represent information flows and *pilot join operations*.

A real example of a query graph is reported in figure 3: it requires a temperature sampling only on the pallets placed in the nearest truck to a given point. The query LLQ1 represents the query that periodically (with period T) samples trucks positions through GPS. HLQ1 is activated every T instants to find the truck nearest to the point P. The ID of the base station mounted on that truck is then inserted in the snapshot *NearestTruck*. The query LLQ2 fills the output stream *Temperatures* and is activated on all the logical objects abstracting temperature sensors and currently connected to the base station with the ID contained in *NearestTruck*.

Figure 4 shows the execution of a query in the system, providing an example of the process of query decomposition and distribution. When a query is sent to the system (Q1), the query analyzer parses it and extracts the definition of the nodes composing the query graph. Needed data structures are instantiated and the execution of High Level Queries is started (Q2). Then, the set of logical objects that will take part in each Low Level Query is identified using a registry. Finally, the

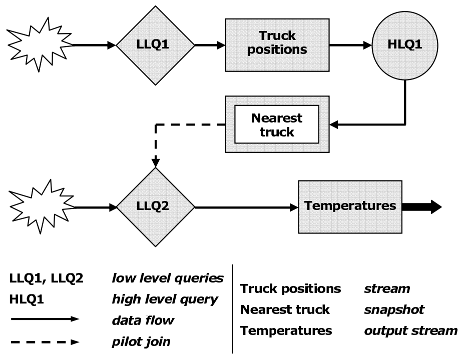

Fig. 3. Query graph example.

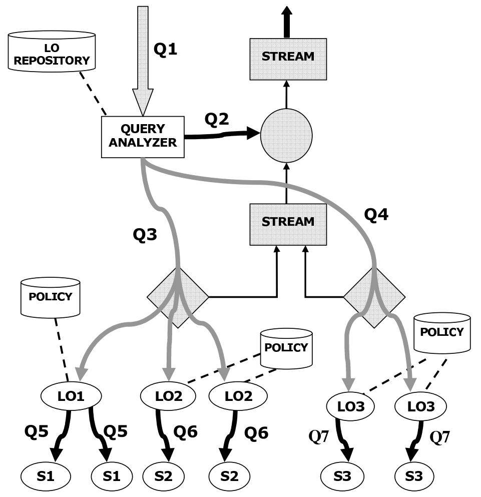

Fig. 4. Query decomposition and distribution process.

system starts sending logical objects the commands needed to execute Low Level Queries (Q3, Q4), that are expressed at a higher abstraction level with respect to the set of commands directly intelligible by physical devices. Abstract requests (Q3, Q4) are translated by logical objects into concrete ones (Q5, Q6, Q7) in order to allow physical devices to execute them. The set of rules used to perform this operation are represented in the figure by the policy container. It is to be noticed that the same abstract command can be translated into different concrete commands if it should be executed on different devices, recognizing different concrete policies constraints. For example, in the figure, Q3 has been translated to Q5 for a class of devices and to Q6 for another class of devices. When all the query components are started, produced data flows go from logical objects to the output data structure, following the path shown in figure 3. If required by the user (with a specific language clause) the set of logical objects participating to the query is updated periodically or when some events happen.

To better define a temporal semantics for the Pilot Join operation, a native timestamp concept was introduced: it is a timestamp field attached to each record produced in the system. Low Level Queries are the first elements of the query graph that generate native timestamp values. As said before, when they are activated, these queries compute some SQL operations on data contained in their local buffer and insert the obtained records in an output data structure. They set the native timestamp field of these records equal to the current timestamp, (i.e. the timestamp at which the query was activated). A synchronization algorithm is used to share the current timestamp among all the system nodes.

Each record generated by High Level Queries is timestamped with the native timestamp of the record that caused the event, if the query is event based, or with the activation timestamp, if the query is activated periodically.

In the following some examples - drawn from the ART DECO case study related to wine production and transport processes - are provided to show the peculiarities of our language with respect to TinyDB and GSN. Suppose that the vineyard is equipped with a set of wireless nodes, having on board a temperature and a humidity sensor. The first query we consider has the role of monitoring environment parameters of the area in which wine is cultivated. More specifically we want to sample temperature and humidity every 30 minutes, returning these values, together with the location of the sensor, *only if the sensed temperature is in a critical range*. In order to provide more accuracy in the results, each node is required to sample its sensors every ten minutes and to consider the average of the three last values. In this situation, the user submitted query is only composed of a Low Level statement:

```
CREATE OUTPUT STREAM EnvironmentParameters
(sensorID ID, temp FLOAT, humidity FLOAT, locationX FLOAT, locationY FLOAT)
AS LOW:
    EVERY 30 m
    SELECT ID, AVG (temp, 30 m), AVG (humidity, 30 m), locationX, locationY
    SAMPLING
        EVERY 10 m
    EXECUTE IF EXISTS (temp) AND is_in_Vineyard(locationX, locationY)
        REFRESH EVERY 5 m
```

Unlike TinyDB, our language can run a unique query simultaneously on different kinds of physical devices, thus supporting runtime heterogeneity. In fact, note that the `Execute If` clause requires that the query be executed on all the nodes currently placed in the yard and having on board a temperature sensor and it does not restrict the query execution to wireless nodes only. Therefore suppose that a worker is in the yard with a PDA which can sense the current temperature: this PDA will execute the query exactly as the wireless nodes (if the PDA has not a humidity sensor on board, null values will be produced). In this example, it should be noticed that the location attribute is static for wireless nodes (and it is configured during the network deployment phase), while it is dynamic for PDAs: if a worker enters the yard his PDA will start executing the query as soon as the *Execute If* clause will be reevaluated, possibly recovering data which should be provided by a "dead" sensor.

Another important peculiarity of our language is the ability of querying the network state as "normal" data. Suppose that, in the previous example, we want to monitor the power state of the wireless nodes in order to detect low powered devices. The following query can be written:

```
CREATE OUTPUT STREAM LowPoweredDevices (sensorID ID)
AS LOW:
    EVERY ONE
    SELECT ID
    SAMPLING EVERY 24 h
        WHERE powerLevel = low
    EXECUTE IF deviceType = "WirelessNode"
```

The previous queries don't make use of the High Level Language which, for instance, is often used when spatial aggregations have to be performed on sampled data. As an example, consider a query that returns the number of low powered wireless nodes: a High Level Query performing the count aggregation can be written on the stream generated by the previous Low Level Query:

```
CREATE OUTPUT STREAM NumberOfLowPoweredDevices (counter INTEGER)
AS HIGH:
    EVERY 24 h
    SELECT COUNT(*)
    FROM LowPoweredDevices(24 h)
```

While the GSN concept of wrapper is quite similar to our Low Level Queries and that of virtual sensor is quite similar to our High Level Queries, the next example shows the pilot join operation that is the feature really improving our language potential with respect to GSN.

We consider the monitoring of wine during the transport. In particular, suppose that every truck is equipped with a GPS and a base station. Suppose also that each pallet has a temperature sensor used to sense the temperature of the contained bottles. The query requires to produce as output the list of pallets whose temperature exceeded a certain threshold while the truck was traveling through a given zone, which is considered particularly critical:

```
CREATE SNAPSHOT TrucksPositions (linkedBaseStationID ID)
WITH DURATION 1 h
AS LOW:
    SELECT linkedBaseStationID
    SAMPLING
        EVERY 1 h
        WHERE is_in_CriticalZone(locationX, locationY)
    EXECUTE IF deviceType = "GPS"

CREATE OUTPUT STREAM OutOfTemperatureRangePallets (palletID ID)
AS LOW:
    EVERY 10 m
    SELECT ID
    SAMPLING EVERY 10 m
        WHERE temp > [threshold]
    PILOT JOIN TrucksPositions
        ON baseStationID = TrucksPositions.linkedBaseStationID
```

In this query the pilot join operation is used to activate the temperature sampling only on the pallets contained in the trucks that are driving into the critical zone. Table II shows the attributes of the logical objects involved in the execution of the query: GPS mounted on trucks and sensors attached to pallets. Note that the interface of logical objects wrapping GPS devices has an attribute "baseStationId" that retrieves the id of the base station mounted on the same truck (this is a static attribute, whose value is defined during the network deployment time).

TABLE II
LOGICAL OBJECTS USED IN THE TRANSPORT MONITORING QUERY

GPS - Logical object wrapping a GPS device			
Field Name	**Data Type**	**Field Type**	**Description**
ID	ID	ID	Logical object identifier
linked BaseStationID	ID	Static	ID of the base station mounted over the truck
locationX	FLOAT	Dyn. prob.	Sensor location X coordinate
locationY	FLOAT	Dyn. prob.	Sensor location Y coordinate
deviceType	STRING	Static	Type of device

WSN node - Logical object wrapping a single WSN node			
Field Name	**Data Type**	**Field Type**	**Description**
ID	ID	ID	Logical object identifier
baseStationID	ID	Dyn. non prob.	ID of the base station the WSN node is currently connected to
temp	FLOAT	Dyn. prob.	Sampled temperature

V. State of the Project and Future Work

The state of the project is the following: we defined the complete non ambiguous EBNF grammar of both Low and High Level Languages, trying to obtain the maximum readability, and we defined the semantics of all language clauses [10]. We are currently implementing the parser and the query analyzer. We plan to initially test the software on dummy logical objects before introducing real physical devices.

From the language definition point of view, some extensions are possible. For instance, an improvement can be the support of actuators, abstracting them in a similar way as sensors. We also evaluated the possibility of introducing clauses for performing in-network data-mining operations, but the first specification of the language does not support them yet.

Acknowledgements

This work is partially supported by the ART DECO FIRB project of the Italian MUR

References

[1] A. S. Tanenbaum, C. Gamage, and B. Crispo, "Taking Sensor Networks from the Lab to the Jungle," *IEEE Computer*, vol. 39, no. 8, pp. 98–100, 2006.

[2] "http://artdeco.elet.polimi.it/."

[3] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[4] ——, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[5] K. Aberer, M. Hauswirth, and A. Salehi, "A middleware for fast and flexible sensor network deployment," *Proceedings of the 32nd international conference on Very large data bases*, pp. 1199–1202, 2006.

[6] ——, "The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks," *TR LSIR-REPORT-2006-006*, pp. 1–21, 2006.

[7] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein, "Entirely declarative sensor network systems," *Proc. VLDB'06*, pp. 1203–1206, 2006.

[8] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network systems," *T.R. UCB/EECS-2006-132*, pp. 1–14, 2006.

[9] L. Baresi, D. Braga, M. Comuzzi, F. Pacifici, and P. Plebani, "A service-based infrastructure for advanced logistics," in *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*. New York, NY, USA: ACM Press, 2007, pp. 47–53.

[10] M. Fortunato and M. Marelli, "Design of a declarative language for pervasive systems," Master's thesis, Politecnico di Milano, 2007.