Sensation: A Middleware Integration Platform for Pervasive Applications in Wireless Sensor Networks

Tilemahos Hasiotis, George Alyfantis, Vassileios Tsetsos, Odysseas Sekkas, Stathes Hadjiefthymiades
Pervasive Computing Research Group, Communication Networks Laboratory,
Department of Informatics and Telecommunications, University of Athens,
Panepistimioupolis, Ilisia, Athens 15784, Greece,
{ thasiot, g.alyfantis, b.tsetsos, o.sekkas, shadj }@di.uoa.gr

Abstract— **In this paper we focus on the issue of application development for wireless sensor networks (WSN). Currently, such networks are extensively used in various business domains. However, their highly customized operating systems and application middleware render the application development for multiple WSNs rather cumbersome. Applications based on multiple WSNs are typical in the emerging pervasive computing paradigm adopted in numerous domains (e.g., telemedicine).** ***A*** **WSN can be considered as a source of information similarly to a database. Motivated by the layered driver approach introduced in the ODBC/JDBC frameworks, we propose a middleware integration architecture. Our architecture presents a unified and developer-friendly interface and abstract data model towards the application. Such interface conceals the peculiarities of the underlying WSNs as their coordination and data retrieval software is integrated in the lower layer of the proposed middleware framework. We present the design of the discussed architecture that is based on open standards like XML.**

I. INTRODUCTION

Recent advances in micro-electro-mechanical systems (MEMS) technology, wireless communications, and digital electronics have enabled the development of low-cost, low-power, multifunctional sensor nodes that are small in size and can communicate over short distances in an ad-hoc manner. Such nodes are, in general, characterized by limited computing and communication capabilities. A large number of such cooperating sensor nodes can be deployed as a wireless sensor network (WSN) [1]. Most modern WSN deployments are targeted at dedicated, proprietary, and specialized applications such as environmental surveillance, habitat monitoring, traffic monitoring, military systems, supply chain management, and healthcare systems [2].

The programming of WSN applications is usually performed through middleware solutions (see Section II) that, in general, interoperate only with specific WSN hardware implementations. Such middleware, contrary to the usual interpretation of this term, does not aim to provide a *generic* and *high-level* WSN programming platform, but rather a basic set of tools and libraries for the *low-level* handling of *technology-specific* sensor nodes. Thus, the programming and configuration of a sensor network are very difficult and error prone tasks, since the application developer has, in the majority of cases, to program individual sensor nodes, using low-level programming languages, directly interfacing with both the hardware and the network modules.

As long as the WSNs are used in specific application domains, this programming approach is well accepted. However, in order to incorporate the WSN infrastructures into other computing paradigms, some enhancements on the existing middleware architectures seem necessary. Nowadays, one of the most promising and pursued computing paradigms is pervasive computing [3], [4] that envisages intelligent networked environments, in which users can seamlessly use context-aware proactive services. In order to bring the pervasive computing paradigm one step closer to its realization, pervasive applications should be capable of seamlessly accessing contextual data, originating from underlying sensing infrastructures. In addition, the applications should be agnostic of the various different underlying middleware technologies and their specific characteristics. Furthermore, in order to make programming in such environments as developer-friendly as possible, there is a need for programming abstractions that simplify the programming process, and for middleware that supports such abstractions [5], [22]. In other words, current and future high-level pervasive applications will be interested in the information itself – not in the particular underlying hardware and networking technologies. These are exactly the applications that the proposed scheme can be integrated with.

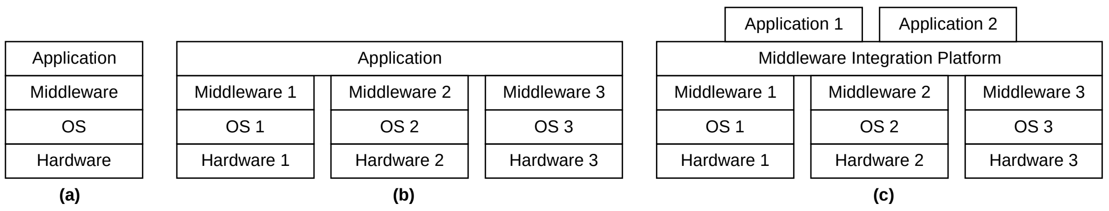

Figure 1. WSN Usage Scenarios. (a) A single application operates over a single WSN (WSN-specific application). This is the case for most of the contemporary applications. (b) A single application uses several different WSNs. The application has to be capable of concealing the heterogeneity. (c) Different applications use several underlying WSNs in a unified manner by means of a middleware integration platform. The applications are agnostic of the underlying technologies (middleware implementations)

From the above, it becomes evident that a WSN application programming framework, in order to fully support the applications of a great variety of users with different needs (characteristics of a pervasive environment), should be designed according to the following principles:

1. It should conceal the heterogeneity between different sensor network infrastructures, in terms of sensor, networking, or middleware technologies. This is necessary, because it is highly unlikely that a single WSN infrastructure will be standardized and will dominate the field (at least in the years ahead).
2. It should provide an “appropriate” programming model (developer-friendly API, information processing facilities, management and reconfiguration procedures) that would further encourage the development of real-world pervasive applications. It is considered that these applications will bring the WSNs to their full potential [6].

The work presented in this paper, describes the design of a middleware integration system that conforms to these two primary architectural principles. Of course, such a middleware framework should also take into consideration the unique characteristics of WSNs: energy scarcity, and the need for robustness and scalability [7], [8]. Moreover, such middleware integration framework paradigm enables competition and flexible business models [27]. Our design is largely motivated by the ODBC/JDBC framework introduced in the database programming community [26]. A sensor network, as a whole, can be considered as an information source similar to a database in the ODBC/JDBC world.

The rest of the paper is organized as follows. We discuss prior work on sensor middleware technologies and identify some of their limitations in Section II. In Section III, we present a high-level middleware integration architecture, which conceals the heterogeneity between different sensor network infrastructures, and provides programmers with an abstract data acquisition mechanism. In Section IV, we discuss the structure of the Unified Sensor Language, which provides seamless communication between the applications and the middleware, enabling clients to pull sensor data from the underlying deployed WSNs. The detailed functionality of the proposed architecture is discussed in Section V. In Section VI, we describe a high-level API, interfacing with the proposed system and capable of performing complex sensing activities in a flexible and intuitive manner. In Section VII we provide our conclusions and future work.

II. EXISTING MIDDLEWARE SYSTEMS

In this Section, we will present some of the most well known work that has been done, over the last few years, regarding the development of middleware systems for WSNs. It has to be stated that we do not propose any improvements for the existing middleware solutions. We neither try to compare them with our framework. Sensation is a middleware integration platform, not really a middleware. Our objective is to give the possibility to different WSN technologies to collaborate and complement one another for the same purpose, i.e., the delivery of sensor data to applications, by means of a software integration layer.

Most applications today are bound to a certain WSN technology, i.e., they are customized in order to interoperate with the WSN of interest (Fig. 1a), making them not portable. In Fig. 1b, we see an extended scenario, where an application is capable of interfacing different WSNs. However, this poses significant difficulties to the application programmer, who must know all the underlying technologies in detail.

TinyDB [9], [10], [23] and Cougar [11], [12], [23] treat the underlying sensor network as a distributed database. They assume a single “virtual” database table, SENSORS, where each column corresponds to a specific type (attribute) of sensor (e.g., temperature, light) or other source of input data (e.g., sensor node identifier, remaining battery power). Reading out the sensors at a node can be regarded as appending a new row to SENSORS. Users specify the data they want to collect through declarative queries, which are expressed in a subset of SQL with some extensions [10] that enable such queries to be periodic and continuous. Both TinyDB and Cougar use a decentralized approach, where each sensor node has its own query processor that preprocesses and aggregates sensor data on their way from the sensor node to the user. Nowadays, TinyDB is considered as one of the de facto standards in middleware technologies; however, the use of declarative (SQL-like) queries does not always constitute a flexible and expressive query mechanism. Moreover, TinyDB does not offer the possibility of defining, on-the-fly, events, but supports only predefined events, programmed into the sensor hardware at compile-time.

SINA [13] is a middleware, which allows sensor applications to issue queries and tasks, collect replies and results, and monitor changes within the network. Its main features include: hierarchical clustering of the sensor nodes and attribute-based naming of the sensor nodes, and a spreadsheet paradigm for organizing sensor data in the nodes. SINA uses SQL-like queries as well as SQTL (Sensor Query and Tasking Language) procedural scripts. SQL-like queries use the aforementioned features to execute simple querying and monitoring tasks, while for more advanced operations, SQTL plays the role of a programming interface between sensor applications and the SINA middleware. An SQTL message, containing a script and encapsulated in a XML-like SQTL wrapper, is meant to be interpreted and executed by the sensor execution environment (SEE), running on each sensor node. Contrary to the TinyDB and Cougar approaches, SINA, by providing an alternative scripting interface, is considered more flexible. However, the actual programming of tasks can be a cumbersome procedure.

SensorWare [14] is another middleware system that abstracts the run-time environment of a sensor node using a set of services, and a scripting language to compose sensing tasks out of these services. The scripting code can migrate from node to node, autonomously. This agent-based approach facilitates monitoring of moving and dynamic phenomena, such as a group of animals moving within a forest. However, this approach, due to its complexity, requires computationally strong sensor nodes. Moreover, the migration of code within the network is a considerable source of energy consumption, and a possible reason for short network lifetime.

Besides the solutions discussed above, there are many other similar middleware approaches. Impala [15], for example, exploits mobile code techniques to change the functionality of middleware executing at a remote sensor. MiLAN [16] is another middleware approach that enables dynamic network configuration (i.e., to identify and organize network resources), in order to provide quality of service guaranties to applications. Finally, DSWare [17] is a middleware approach based purely on the notion of events. The application declares interest in certain state changes of the real world and specifies the corresponding course of action.

As already stated, the above middleware systems follow different programming paradigms (e.g., database approach, agent-based approach, event-based approach) and differ with respect to ease of use, expressiveness, scalability, and overhead. What has to be noted is that the application developer is tightly coupled with these middleware systems, and must have in mind their specific characteristics, capabilities and limitations. Moreover, he has to be familiar with the corresponding specific programming interfaces. However, in order to make the development of WSN-based pervasive applications a reality, the application developer should be unaware of the underlying middleware technologies and their characteristics.

The system, discussed in the next Sections, aims to facilitate the development of pervasive applications by focusing on the following goals (see also Section I):

1. Support for heterogeneity of the existing (and future WSN-specific middleware) through an abstraction layer.
2. A high-level and intuitive programming model for context-aware pervasive applications.

Such objectives could be achieved with an architecture of the form of Fig. 1c. Contrary to Figures 1a and 1b, applications are agnostic of the underlying technologies and access sensor data in a unified manner, by means of a middleware integration framework.

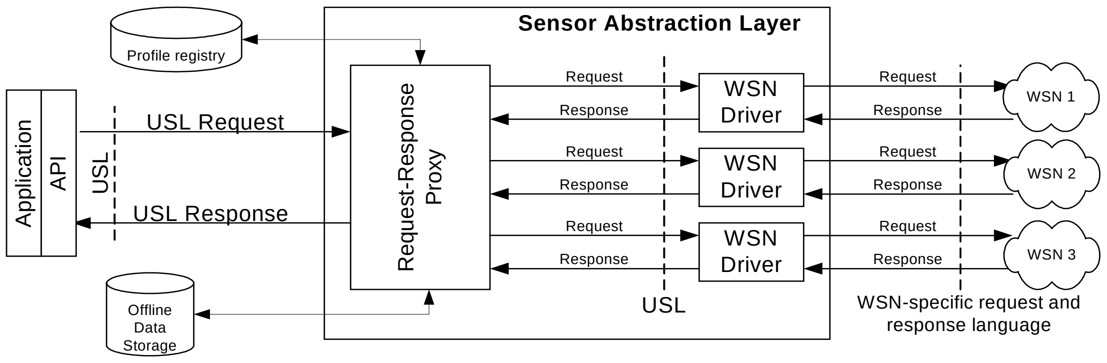

Figure 2. General System Architecture

III. Architecture Overview

In this Section, we discuss how the goals referenced in the previous Section can be achieved. In order to conceal the heterogeneity between different sensor network infrastructures (in terms of sensor, networking, or middleware technologies) an intermediate layer has to be added, which will facilitate the communication of monitoring (and various other) applications with such sensor networks.

This layer will serve as an abstraction layer hiding the different WSN implementations from end-user applications. Requests and responses from various heterogeneous networks will be performed in a unified way, with the aid of a well-defined language. We call this language USL after the Unified Sensor Language. Applications will have to be aware of this language (Fig. 2) and comply with it. Requests arriving at the Sensor Abstraction Layer (SAL) have to be analyzed, then translated to the corresponding WSN-specific languages, and forwarded to the underlying WSNs. Conversely, replies from the various WSNs, following a proprietary format, have to be translated to USL, aggregated and sent back to the requestor. This layer can be considered as the core of the proposed system. We chose XML [18] for the implementation of USL, as it is a standardized and widely used meta-language, capable of describing data structures in a formal and structured manner. Moreover, the automated processing-transformation of XML-based syntaxes can facilitate the discussed functionality.

Regarding the posed objective of providing application programmers with a high-level and intuitive data programming model, an efficient programming interface has to be overlaid to the aforementioned SAL, as shown in Fig. 2. This will be an Application Programming Interface (API), providing programmers with expressive commands to flexibly perform a variety of requests to the sensor infrastructure. This API will be implemented in different levels of abstraction (versions) and points of view of the environment, that a programmer could use depending on his specific needs – e.g., a low-level version of the API would provide the programmer with great flexibility, but it will need more elaborate programming, while a higher-level one would be much friendlier, but possibly with limited flexibility, as it will be discussed in Section VI.

Our current design of the API is based upon the location and the device abstraction. The programmer can consider the environment as a combination of different sensor-enabled locations and devices. Contrary to locations, devices are not considered stationary. They can be portable gadgets (e.g., PDAs) enriched with sensing capabilities. Other entities, such as persons, robots, or vehicles, can carry such devices, and can be semantically associated with them. In that way, applications can request sensor information on certain areas and/or entities that are associated with a sensor-enabled device.

As shown in Fig. 2, the system is also supported by a *Profile Registry* and an *Offline-Data Storage* (ODS) facility. The former is comprised of several database tables and configuration files and keeps information about the configuration, the supported capabilities and features of the underlying WSNs – e.g., the types of deployed sensors (temperature, humidity sensors, etc.), the measurement units (Celsius, Fahrenheit, Volt, etc), basic functions that transform raw sensor data to more usable formats (e.g. voltage to temperature converter), as well as information about the WSN-specific gateways. All this information is quite necessary and is used by the SAL in order to discover the existing WSNs that can satisfy the submitted requests and eventually forward these requests to them. Moreover, the Profile Registry stores information about the portable and sensor-enabled devices. Whenever the deployed WSNs are altered, in any way, the system administrator should also update the corresponding settings in the Profile Registry.

The ODS is based on a common database system (e.g., relational database) and is responsible for intercepting and storing every response passing through the SAL, in order to make it available for users interested in the history and the statistical properties of the phenomenon they observe. The ODS facility will enable complex information processing (e.g., post-processing for mining stream data and caching), which is a very active research area today. The design of the ODS Schema as well as the detailed description of the Profile Registry are still in progress and thus will not be further discussed in this paper. The structure of the SAL will be further studied in Section V. In the next Section we discuss the definition of the USL interface.

IV. THE UNIFIED SENSOR LANGUAGE

The Sensor Abstraction Layer (SAL), as described in Section III, is based on USL, a generic sensor handling language. This language has been defined after extensive requirements analysis, during which, we collected both the requirements for the interaction between the developer and the sensor-enabled environment/devices and the actual functionality of the currently available WSN middleware systems (see Section II). The analysis resulted in the following requirements for such language:

1. Support for synchronous requests (queries) that retrieve the requested data, either from the WSN or the ODS, and return the corresponding responses in real-time.
2. Support for event-driven programming. Many context-aware applications need to trigger some actions after some events have been generated from the WSN. The users should be able to register listeners and handlers of such events (e.g., in case the temperature is higher than 42 ^{o}C, and indoor light is very bright, the fire alarm should be activated).
3. Support for periodic monitoring of the sensor values (e.g., sensing of the temperature every 5 minutes, starting from Monday 2 July, 15:00 p.m. and stopping after a week).
4. Easy and dynamic change of the supported sensor types and sensor functions. As new WSNs or sensors are deployed, the programmer should be able to include the new functionality in his program logic. Moreover, in case of sensor failures, which is a common case, adequate error handling should be enabled by descriptive error indications.

The final outcome of the requirements analysis was the USL language specification. USL is described in XML format, since XML provides platform independence, interoperability and can be easily parsed [18]. Given the recent popularity of the database-like information processing for sensor network data [9], [11], one could claim that an SQL-like language would be more suitable. We believe that the relational nature of the SQL-like languages [25] poses some limitations to the generic handling of the WSN infrastructures, as it is coupled with the concept of relational queries (select-from-where clauses) and does not support very declaratively the event-driven nature of the user interactions. Furthermore, we intend to enhance USL with management functionality (e.g., support for policies, administration and reconfiguration of the WSN), which would be hindered significantly by the adoption of a SQL-like grammar. USL defines two main entities, the Request and the Response. Abstracted versions (due to space limitations) of the XML Schemata [19] of these two entities are shown in Fig. 3 and Fig. 4. The complete XML Schemata can be found in [28].

A. *The USL Request*

The USL Request represents the requests (synchronous, event-driven and periodic) of the user towards the WSN or the ODS. The root element *Request* contains a unique ID attribute. This ID is generated by the API (see Section VI) and uniquely identifies a client request. In case of a synchronous query, the client specifies the contextual information it is interested in within the *RequestedInfo* element. This information may be one or more sensor readings, the location where a condition holds, the time instance or the duration of a specific phenomenon, the device ID that satisfies some criteria or a combination of the above. Of course, not all combinations are considered valid, so in the WSN Driver a semantic checking procedure is performed (see Section V). The constraints of the query are described in the *QueryFilter* element. In this element, one can declare time conditions (*TimerExpr* element), sensor conditions (*SensorExpr* element) and limit the query to a specified location or sensor-enabled device (*Location* and *Device* elements respectively). The *SensorExprType* (see Fig. 3) is the XML representation of the following Extended BNF [20] grammar:

SensorExpr = [Function,] SensorType, Conditional, Value;

Function = ‘tempAverage’ | ‘tempMinimum’ | ‘tempCount’ |‘tempMaximum’ | ‘spatialAverage’ | ‘spatialSum’;

SensorType = ‘Temperature’ | ‘Humidity’ | ‘Acceleration’;

Conditional = ‘Greater’ | ‘Less’ | ‘Equals’ | ‘WithinRange’;

Value = Alphanumeric | RealNumber | Integer;

Of course, the definitions of *Function* and *SensorType* are not complete. Their potential values are defined in a separate XML Schema described at the end of Section IV.B. The *TimerExpr* element has a similar syntax.

Additionally, there is an optional *GroupBy* sub-element that can group the results similarly to the known SQL functionality and has almost the same syntax as *RequestedInfo*. The *Monitor* element, if present, denotes that the query should be executed periodically as described in the *StartTime*, *StopTime* and *Period* elements. From now on, we will refer to these periodic queries as *monitors*. Similarly to the events, they require that the developer register listeners accordingly.

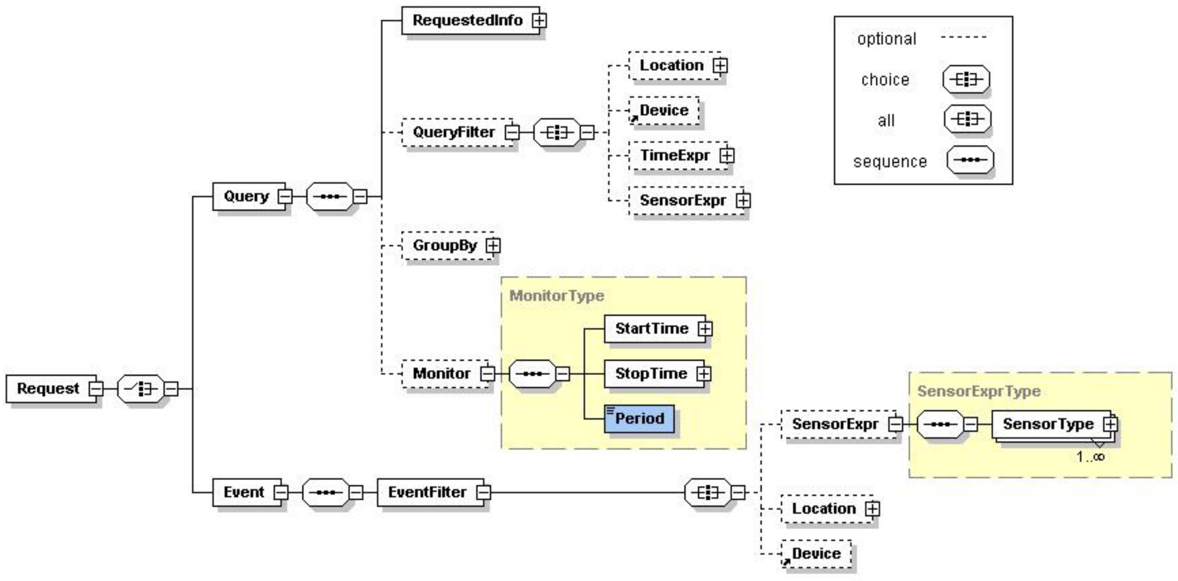

Figure 3. The XML Schema of the USL Request

On the other hand, in case of event-driven programming, the only sub-element of *Request* is the *Event*. This element contains a set of conditions (*EventFilter* element) that, when satisfied, trigger some events to the upper layers of the middleware framework. These upper layers have already registered listeners for these events and upon receipt of an event (through the USL Response entity) they perform some predefined (by the developers) actions. The *EventFilter* is very similar to the *QueryFilter* except for the fact that it does not contain time conditions. This seems quite reasonable for this first version of USL as events generated from time conditions can only be implemented with the aid of the ODS and require complex information processing techniques. As there is ongoing research in these areas [21], a future version of USL may also support time-based events (e.g., if the temperature change rate in a computer room is +3 oC/h, inform the building caretaker so as to check the air conditioner).

The USL Request, apart from enabling the registration of event-listeners and the description of queries, can also dispose the already registered event-listeners or monitors. For that purpose the *Event* and *Monitor* elements contain the boolean attribute abort. When a user disposes an event or monitor, its known ID is passed in the ID attribute of the *Request* element and the corresponding *abort* attribute is set to *true*, while all the other elements are absent or blank.

B. The USL Response

The USL Response is much simpler than the Request entity. The root element *Response* has also an ID attribute, and always contains the *ReturnStatus* element. If the error attribute of this element is set to true, then an error has occurred within the SAL or the WSN and its type (*ErrorType* element) is returned to the API in order to raise an application exception. If no error occurred and the request defined a query (or a monitor), the requested data (in *RequestedInfo*) are returned to the requestor. Alternatively, if the request registered an event, then all the elements except for the *ReturnStatus* are absent (i.e., the response is equivalent to a flag indicating that the event has taken place).

Some of the parameters in the aforementioned USL elements and attributes may vary occasionally (e.g., due to deployment of new sensors). These are the sensor types, the (unit transformation) functions that can be applied on them and the types of the error indications. All these are described as enumerations in a separate XML Schema document and are included in the above Schemata in order to impose some constraints during the XML validation of the USL request/responses by the RR Proxy. This separate XML Schema can be regarded as part of the Profile Registry, because it is updated whenever the configuration of the WSNs is modified.

This makes the framework more flexible and WSN independent. Specifically, Sensation can provide to the upper-layer API any functionality that an underlying WSN (or a combination of WSNs) could supposedly support. It is anticipated that in the future there will be geographically overlapping WSNs with different capabilities. In such an environment, Sensation could creatively combine information from complementary WSNs. If a particular WSN does not support a given functionality, the proposed framework provides sound mechanisms for error notification and handling.

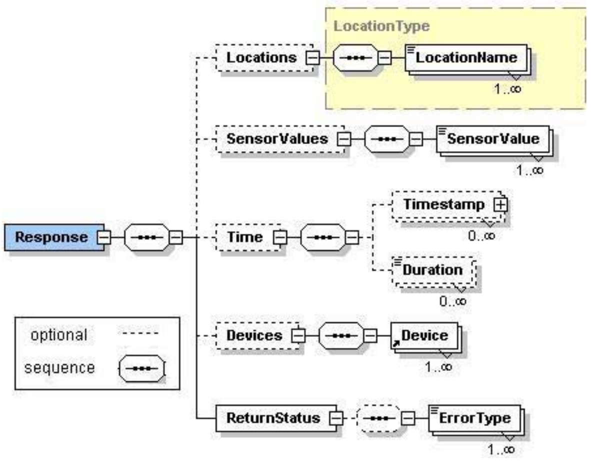

Figure 4. The XML Schema of the USL Response

The formation of a USL Request can be performed by the API discussed in Section VI, and its handling by the SAL. Conversely, a USL Response is filled with the retrieved (or aggregated) data or the possible error indications by the SAL (see Section V), and finally, returned to the client.

V. THE SENSOR ABSTRACTION LAYER

As mentioned in Section III, the SAL is the most important part of the system, and is composed of the following functional entities (see Fig. 2):

- The RR Proxy (Request-Response Proxy), and
- The WSN Driver (one or more, depending on the different deployed WSNs)

A. The RR Proxy (Request-Response Proxy)

The RR Proxy, as shown in Fig. 2, receives the USL requests and performs a first level processing, involving the following steps (Fig. 5):

- Firstly, it checks if the request complies with the USL grammar, i.e., the XML-based document describing the request is validated against the XML Schema (see Section IV). If the validation succeeds, the processing of the request continues, otherwise an appropriate error indication (e.g., Not Valid Request) is returned to the client.
- After a successful validation, the RR Proxy obtains the available WSNs in the specified location or device from the Profile Registry. If the available WSNs are more than one, then the RR Proxy has to dispatch the request to the corresponding WSN Drivers. The RR Proxy also registers a listener for this request, which is associated with the specific request ID that was assigned to the request by the client API. It also has an associated timer that expires after a specified time period. The listener aggregates the responses that are associated with a certain request ID and were returned by the underlying WSNs (or the ODS facility) and returns the aggregated data to the client. If no response is received within the specified timeout interval, the listener returns to the client an error indication (e.g., Request Timeout). The main request processing takes place within the WSN Driver, which is subsequently described.

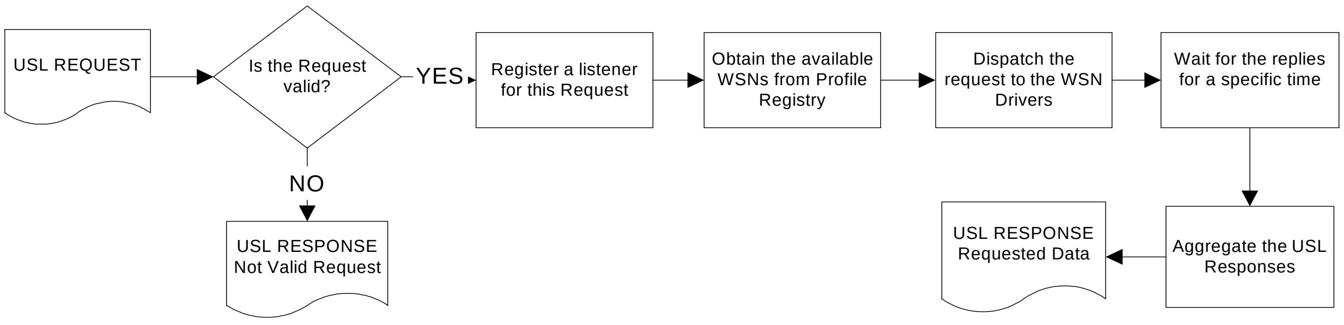
USL REQUEST
Is the Request valid?
YES
NO
Register a listener for this Request
Obtain the available WSNs from Profile Registry
Dispatch the request to the WSN Drivers
Wait for the replies for a specific time
USL RESPONSE Not Valid Request
Aggregate the USL Responses
USL RESPONSE Requested Data

Figure 5. RR Proxy functionality

B. The WSN Driver

As shown in Fig. 6, the WSN Driver processes a request received by the RR Proxy by involving the following steps:

1) USL Request Parsing (Syntactic Analysis): The WSN Driver interprets the data elements of the XML-based USL request.

2) Semantic Analysis:

a) Type Checking: After parsing the USL request, the WSN Driver has to examine, whether the underlying WSN, for which it is responsible, is capable of supporting the request. Firstly, it extracts the list of sensor types (e.g., temperature) to be queried from the RequestedInfo element of the USL Request, and the sensor types (attributes) that appear in the filters from the QueryFilter (or EventFilter) of the USL request. Then, it consults the Profile Registry for the sensor types supported by the underlying WSN, and in case that any of the attributes of the QueryFilter (or EventFilter) is not supported, the processing ends and an error indication (e.g., Not Supported Request) is returned to the RR Proxy. Otherwise, if all the attributes of the QueryFilter (or EventFilter) and at least one of the requested attributes are supported then the processing goes on.

b) Semantic Checking: During this step the WSN Driver checks the semantic validity of the request. If the semantic check fails, an error indication (e.g., Not Valid Request) is returned to the RR Proxy and the whole process ends.

c) Request Routing: If a time expression is among the requested data (e.g., "when was the temperature over 50^{o}C?"), the processing pertains to offline data, and it is served by the ODS facility. Otherwise, the request will be served by the WSN.

3) Request Code Generation: This step relies on the decision made on the previous step. If the request has to be processed offline, then the USL request will be translated to an SQL query, else, if the request has to be processed in real time by the sensor network, the USL request will be translated to WSN-specific code.

4) Request Injection: If all previous steps were completed successfully, the resulting request is injected into the underlying WSN or the ODS facility and the WSN Driver waits for a reply.

The steps followed by the WSN Driver for the formation of the response (Fig. 7) are:

1) Data Collection and Response Formation: The WSN Driver receives a reply from the WSN or the ODS facility and converts it to a USL Response complying with the structure described in Section IV.

2) ODS Update: Before returning the response to the RR Proxy, a tuple will be inserted in the ODS facility (only in case of a real-time request to the actual WSN). This row will contain the reply returned by the WSN, the request ID and additional information (e.g., a timestamp, the location or device). By logging these results, one can later access them for statistical or other kind of processing (i.e., post-processing of sensor data).

3) Response Forwarding: The USL Response is returned to the RR Proxy and, more precisely, to the listener that was registered for the specific request.

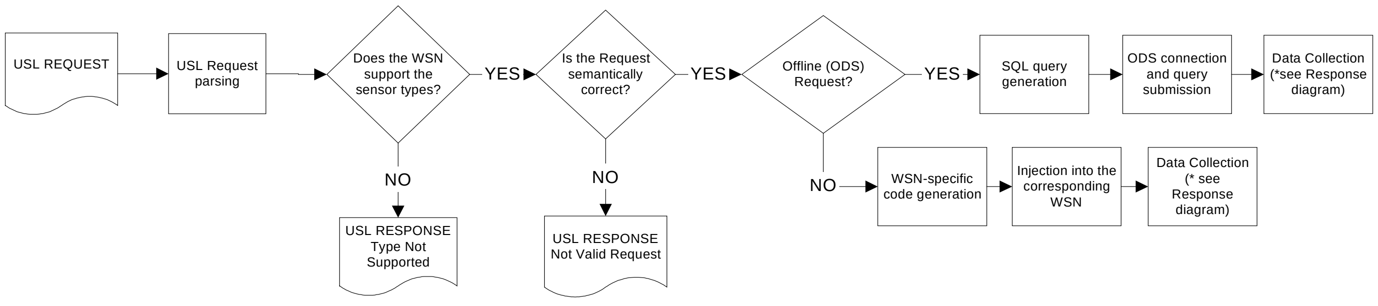
USL REQUEST
USL Request parsing
Does the WSN support the sensor types?
YES
NO
USL RESPONSE Type Not Supported
Is the Request semantically correct?
YES
NO
USL RESPONSE Not Valid Request
Offline (ODS) Request?
YES
NO
SQL query generation
ODS connection and query submission
Data Collection (*see Response diagram)
WSN-specific code generation
Injection into the corresponding WSN
Data Collection (* see Response diagram)

Figure 6.WSN Driver functionality (Request)

What has to be noted is that the communication between the RR Proxy and the WSN Drivers, i.e., within the SAL, is based on the USL. This allows the SAL to be deployed in an implementation independent (language neutral) manner. Furthermore, with the proposed architecture, a WSN manufacturer, providing a WSN Driver for his specific WSN (e.g., a component capable of transforming

USL to WSN-specific language and vice versa), can plug his WSN into the discussed architecture and make it available seamlessly to already existing applications (the term “driver” came up from this vision of WSN software components that will have the same functionality as the common drivers for hardware devices and peripherals, e.g., printers).

For the actual deployment of the infrastructure, the manufacturer has also to configure the Profile Registry with information regarding the WSN capabilities and the functions for the conversion of raw sensor data to best understood formats (e.g., Volt to Celsius degrees). In the following Section we discuss the definition of the Java-based API, overlaid to the USL interface.

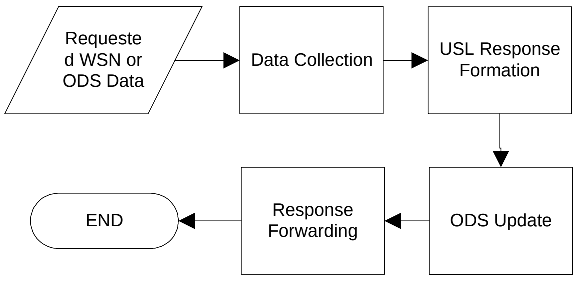

Figure 7. WSN Driver functionality (Response)

VI. WSN Application Programming Interface

As already mentioned in Section III, one of the basic objectives of the proposed middleware framework is the development of a high-level Application Programming Interface (API). Such API (which is currently comprised of two sets of API methods), implemented in Java, will enable programmers of pervasive applications to exploit the functionality and capabilities of the underlying WSNs by providing a set of well-defined methods for requesting and carrying out lower-level WSN services (i.e., sensor data retrieval).

We designed two different APIs, which serve different purposes: one that is based on the concepts of location and device of interest (high-level API), and one that is based on the concept of queries (low-level API). Application developers are capable of using either of them, according to their needs. Some indicative methods of the APIs are presented in Table I. The first API is more developer-friendly and intuitive, while the second is a lower-level, yet more flexible, interface. Its flexibility lies in the fact that users can construct more complex requests, decoupled from the location and device concepts that can sometimes prove a restrictive factor. Notice that the location-centric API incorporates certain location management methods, but these do not fall within the scope of the present paper.

TABLE I. An abstract version of the basic API methods

	Method
High-Level API	**Location.getSensorValue(SensorType, Function)** Returns the requested (function of the) value of the given sensor type, in the given location. The function can be *max, avg, etc.* *
	Location.getDeviceWhere (SensorConditions) Returns the device ID that satisfies the given sensor conditions.
	Location.getTimeWhen (SensorConditions)** Returns a timestamp that satisfies the given sensor conditions.
	Location.addListener(EventFilter) Registers a new listener for a given location. It is triggered when the conditions specified in the EventFilter are satisfied.
	Device.getSensorValue (SensorType, Function) Similar to the first method of this table.
	Device.getLocation () Returns the location of the given device

Low-Level API	**Query.setSensorTypes (SensorTypes)** Sets the requested sensor types (e.g., temperature, humidity, etc.)
	Query.setFunctions (Functions) Sets the functions (*min ,max, etc.*). These should correspond one-by-one to the sensor types of the previous method.
	Query.setMonitor(Monitor) Indicates that the query is monitor. The monitor attribute is a Monitor object that contains the time parameters.
	Query.abort() Cancels the query represented by the query object.
	Query.send() Sends the constructed query to the RR Proxy.
Metadata Methods	**getSupportedSensorTypes(Location)** Returns the sensor types supported by the system in a given location.
	getSupportedTemporalFunctions() Returns the temporal functions supported (e.g., min, avg, count, etc.)

* the functions that can be applied on a sensor type are categorized in spatial and temporal functions

** the presence of "time" in a method (either in its name or the attribute list) denotes offline processing by the ODS

A problem that emerges when encountering dynamically changing heterogeneous networks is the developer's ignorance of the supported sensors types and their functions. The developer is in need of a mechanism to know which functionality is supported by the underlying infrastructure. To overcome this barrier the integration of JDBC-like metadata classes seemed appropriate [24]. By examining metadata information from the *Profile Registry*, applications can discover and use only the available sensor types and functions. Indicative metadata methods are presented in Table I. Metadata should be updated at the occurrence of a change in the deployment of an underlying WSN, e.g. when a new sensor type (temperature, velocity, etc.) is installed, or a supported sensor type has been removed.

VII. CONCLUSIONS & FUTURE WORK

In this paper we discussed a middleware integration platform solution for pervasive applications in wireless sensor networks. The name of the proposed middleware platform is Sensation. The main objective of the architecture is to conceal the heterogeneity between different sensor network infrastructures attributed to the variety of sensor, networking, or middleware technologies. Moreover, it is designed in order to provide a developer-friendly programming model (i.e., an API with information processing facilities, management and reconfiguration procedures) that can further encourage the development of pervasive applications. Furthermore, it is considered that these applications will bring the WSNs to their full potential, as they will penetrate our everyday lives.

In the future, after evaluating our architecture with a TinyDB-enabled WSN, we plan to focus our study on information management issues on WSNs. Specifically, we intend to provide a formal schema of the ODS, which would be most suitable for the processing of stored sensor data, and complex inference procedures. Moreover, we plan to study and develop lower level information processing techniques, in order to cope with the known problem of noisy (or even missing) sensor readings. This will aim to the identification of possibly malfunctioning nodes, which would be able to self-heal, if they are smart enough; otherwise such techniques would help a management entity to single out, or calibrate, incorrect sensor readings, before they reach the requesting client.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "*A Survey on Sensor Networks*", IEEE Communications Magazine, August 2002

[2] K. Martinez, J. K. Hart, and R. Ong, "*Sensor Network Applications*", IEEE Computer, August 2004

[3] M. Weiser, "*The computer for the twenty-first century*" Scientific American, September 1991. (Reprinted in IEEE Pervasive Computing, Jan-Mar 2002)

[4] M. Satyanarayanan, "*Pervasive Computing: Vision and Challenges*" IEEE Personal Communications, August 2001.

[5] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "*The Emergence of Networking Abstractions and Techniques in TinyOS* ", NSDI, 2004

[6] D. Estrin, D. Culler and K. Pister, and G. Sukhatme, "*Connecting the Physical World with Pervasive Networks*", IEEE Pervasive Computing, January-March 2002

[7] K. Romer, O. Kasten, and F. Mattern, “*Middleware Challenges for Wireless Sensor Networks*”, Mobile Computing and Communications Review, Volume 6, Number 2, October 2002

[8] K. Romer, “*Programming Paradigms and Middleware for Sensor Networks*”, GI/ITG Fachgespräch Sensornetze, Karlsruhe, February 2004

[9] S. R. Madden, J. Hellerstein, and W. Hong, “*TinyDB: In-Network Query Processing in TinyOS*”, Version 0.4, September, 2003

[10] S. R. Madden, “*The Design and Evaluation of a Query Processing Architecture for Sensor Networks*”, Ph.D. Thesis. UC Berkeley. Fall, 2003 (TinyDB)

[11] P. Bonnet, J. E. Gehrke, and P. Seshadri, "*Querying the Physical World*", IEEE Personal Communications, vol. 7, no. 5, pp 10-15, Oct. 2000. (Cougar)

[12] Y. Yao, and J. E. Gehrke, “*The Cougar Approach to In-Network Query Processing in Sensor Networks*”, Sigmod Record, Volume 31, Number 3, September 2002.

[13] C.-C. Shen, Ch. Srisathapornphat, and Ch. Jaikaeo, “*Sensor Information Networking Architecture and Applications*”, IEEE Personal Communications, August 2001 (SINA)

[14] A. Boulis and M. B. Srivastava, "*A Framework for Efficient and Programmable Sensor Networks*", In proceedings of OPENARCH 2002, New York, June, 2002.

[15] T. L. and M. Martonosi. "*Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems*". ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, June 2003

[16] W. Heinzelman, A. Murphy, H. Carvalho and M. Perillo,"*Middleware to Support Sensor Network Applications*", IEEE Network Magazine Special Issue. Jan. 2004 (MiLAN)

[17] Sh. Li A1, S. H. Son A1, J. A. Stankovic, “*Event Detection Services Using Data Service Middleware in Distributed Sensor Networks*”, Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Volume 2634 / 2003

[18] W3C Architecture Domain, Extensible Markup Language (XML), www.w3.org/XML/

[19] W3C Architecture Domain, XML Schema, www.w3.org/XML/Schema

[20] D.H.Crocker, "*Standard for the format of ARPA Internet text messages*", STD11, RFC 822, UDEL, August 1982

[21] F. Z., and L. Guibas, “*Wireless Sensor Networks: An Information Processing Approach (Morgan Kaufmann Series in Networking)*”, Morgan Kaufmann, July 2004

[22] J.Hill et al., “*System Architecture Directions for Networked Sensors,*” Proc. Int’l Conf. Arcitectural Support for Programming Languages and Operating Systems (ASPLOS), ACM Press, 2000, pp. 93-104

[23] J.Gehrke, and A. Madden. “Query Processing in Sensor Networks,” IEEE Pervasive Computing, January-March 2004

[24] M. Fisher, J. Ellis, and J. Bruce, “*JDBC API Tutorial and Reference, Third Edition*”, Addison-Wesley Pub Co, 2003

[25] C.J. Date, “*An Introduction to Database Systems, Eighth Edition*”, Pearson Education, 2003

[26] R.Orfali, D.Harkey, “*Client/Service Programming with Java and CORBA*”, 2nd edition Wiley, 1998

[27] V. Tsetsos, G. Alyfantis, T. Hasiotis, O. Sekkas, and S. Hadjiefthymiades, “*Commercial Wireless Sensor Networks: Technical and Business Issues*”, to appear in the proceedings of WONS (Wireless On demand Network Systems and Services) 2005 Conference

[28] P-comp Research Group, Sensation Platform, http://p-comp.di.uoa.gr/sensation

APPENDIX I. AN EXAMPLE REQUEST-RESPONSE OPERATION

In this appendix we intend to give some insight to the whole process of issuing a request against the sensor nodes and retrieving their readings. Consider a scenario where the patients in a hospital are equipped with sensors from different vendors, one measuring the heartbeat, another one the temperature of the body, a third the blood pressure, etc. Doctors and nurses are interested in receiving information on the health status of every patient, and are not aware of the different sensor technologies that are used. They only know which sensors are attached to each patient. The proposed architecture is capable of offering this level of abstraction.

Suppose a health management application needs to know the heartbeat rate and the body temperature of a patient named “John Malade”. The application needs firstly to query the hospital’s database, in order to find out which sensors are attached to Mr. Malade, and then single out the sensors that are of interest (i.e., heartbeat rate and the body temperature sensor). Consider that heartbeat sensor #57, and temperature sensor #309 are attached to the patient in question. The corresponding entries in the profile registry, describing these two distinct devices, would look like Heatbeat_57, and Temperature_309. With this information the application is capable of retrieving the patient’s status. The corresponding Java code (see Section VI) should look like this:

```
Device heartBeat = new Device(“Heatbeat_57”);
//creates an instance of the heartbeat sensor node
Device temperature = new Device(“Temperature_309”);
//creates an instance of the body temperature sensor node
```

```
String[] heartSupportedSensors = MetaData.getSupportedSensorTypes(heartBeat);
//the programmer is informed about the supported sensor types.
String[] tempSupportedSensors = MetaData.getSupportedSensorTypes(temperature);
//the programmer is informed about the supported sensor types.

SensorType[] sensorTypes = new ArrayList();
sensorTypes.add(heartSupportedSensors[0]);
//we declare interest for the heartbeat rate
String[] results = heartBeat.getSensorValues(sensorTypes);

sensorTypes.clear();
sensorTypes.add(tempSupportedSensors[0]);
//we declare interest for the body temperature
String[] results = temperature.getSensorValues(sensorTypes);
```

The code provided previously generates two different requests, one for the heartbeat, and one for the temperature, since these two sensors are of different technologies, and, thus, cannot belong to the same WSN. As previously stated, this requests, expressed by the WSN API, will subsequently be transformed to USL (i.e., XML documents) and be assigned unique IDs (e.g., 123456, and 123457). Below, we present only one of the two requests (e.g., the body temperature portion). The USL Request document should look like this:

```
<Request xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=" WSNRequest.xsd"
ID="123456">
   <Query>
      <RequestedInfo location=”false” time-instance=”false” device=”true” sensor=”true” time-duration=”false”>
         <SensorList>
            <SensorType function=”NoFunction” type=”temperature”/>
         </SensorList>
      </RequestedInfo>
      <QueryFilter>
         <Device ID=”543”/>
      </QueryFilter>
   </Query>
</Request>
```

This XML-formatted request is passed to the RR-proxy that checks its syntactic validity and registers a corresponding listener bound to the request ID. Subsequently, the USL Request is forwarded to the underlying WSN-driver (i.e., the TinyDB compliant driver in our case). There, it will be translated to a TinyDB (SQL-like) query that should look like this:

```
SELECT “Temperature”
   FROM SENSORS
   WHERE MoteID=”534”
```

Subsequently, this query will be sent to the corresponding TinyDB WSN gateway and will be injected into the sensor network. When the response from the mote with ID=534 (i.e. “Temperature_309” – Mr. Malade’s thermometer) reaches the gateway, it will be sent back to the WSN-driver as a raw data record. Eventually, this response will be translated to a USL Response XML document, which should look like this:

```
<Response xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="
WSNResponse.xsd" ID="123456">
   <SensorValues>
      <SensorValue function=”NoFunction” type=”temperature”>37</SensorValue>
   </SensorValues >
   <ReturnStatus error=”false”>
   </ReturnStatus>
</Response>
```

This USL Response arrives at the RR-proxy, where there is an appropriate listener, and finally returns to the WSN API (and, thus, to the management application), since no errors have occurred.