PerLa: A Language and Middleware Architecture for Data Management and Integration in Pervasive Information Systems

Fabio A. Schreiber, *Life Senior Member*, *IEEE*, Romolo Camplani, Marco Fortunato, Marco Marelli, and Guido Rota

Abstract—A declarative SQL-like language and a middleware infrastructure are presented for collecting data from different nodes of a pervasive system. Data management is performed by hiding the complexity due to the large underlying heterogeneity of devices, which can span from passive RFID(s) to ad hoc sensor boards to portable computers. An important feature of the presented middleware is to make the integration of new device types in the system easy through the use of device self-description. Two case studies are described for PerLa usage, and a survey is made for comparing our approach with other projects in the area.

Index Terms—Declarative language, device heterogeneity, functionality proxy, middleware infrastructure, pervasive system, SQL, wireless sensor networks.

1 Introduction

In recent years, the number of real applications requiring the cooperation among several heterogeneous devices has been rapidly increasing. Components belonging to very different technologies and characterized by different complexity levels, ranging from simple RFID tags to on-board PCs, are often employed together within the same scenario. The application goals can be various, but they can usually be reduced to the gathering of raw data from the environment, the processing of collected data, and the control of some environmental parameters.

Home automation and monitoring of natural catastrophic phenomena, like earthquakes and rockfalls, are two completely different examples of applications interacting with many heterogeneous devices. Although the goals of these applications and the types of sensors and actuators involved are completely different, the issues to be managed are exactly the same. First, data have to be sampled from several heterogeneous nodes belonging to different technologies and having different sampling semantics; then, gathered data have to be integrated and processed in order to extract information relative to the monitored environment; finally, the obtained information can be used to control a set of actuators able to modify some environmental conditions or to report alarm messages.

• *The authors are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Via Ponzio 34/5, Milano 20133, Italy. E-mail: {schreiber, camplani}@elet.polimi.it, fortunato_marco@alice.it, marelli_marco@hotmail.com, guido.rota@gmail.com.*

Manuscript received 3 Nov. 2010; revised 4 Feb. 2011; accepted 12 Feb. 2011; published online 2 Mar. 2011.
Recommended for acceptance by N. Medvidovic.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-11-0328.
Digital Object Identifier no. 10.1109/TSE.2011.25.

As an example, consider one of the case studies of the *ARTDECO* project [1], related to the monitoring of the wine production process from the vineyard to the table. Many hardware technologies are employed in order to gather all the required data: Ad hoc boards hosting temperature, humidity, and pressure sensors are placed in the vineyard; RFID tags and GPS devices provide tracking support during the transport phase; Personal Digital Assistants (PDAs) allow human operators working on the vineyard to interact with the system. Some actuators, able to modify the temperature and the humidity levels in the yard, are also provided in order to allow environmental control when monitored parameters are out of their optimal ranges.

The implementation of a system able to achieve the above-presented application goals usually implies the management of many ad hoc heterogeneous embedded devices in order to sample all the environmental variables involved. Each of these devices can belong to different technologies and can be characterized by different features, both in terms of computational power, power supply requirements, and exposed Application Program Interfaces (APIs). Moreover, in some applications presenting unusual requirements, the design of specific hardware platforms becomes a necessary task, increasing in this way the overall heterogeneity level. As a consequence, the development of an ad hoc software infrastructure for each device and for each application to be deployed is the easiest and the most common approach. However, it has the main disadvantage of requiring the recoding of large pieces of software whenever the application requirements slightly change or a new device has to be integrated within the existing system.

Splitting the whole problem into two different activities is certainly a better approach: First, a data gathering phase handles sampling issues; then, a data processing task

extracts information from sampled data. Many research efforts are nowadays related to the second phase and many of the academic projects currently under development are mostly focused on the information extraction issues rather than on the data collection ones. More specifically, as we discuss in Section 2, there are a number of projects aiming at developing software components, offering autonomic services able to self-organize and self-adapt in order to provide adaptive and situated communication services. These works try to identify a fundamental and uniform abstraction, able to characterize autonomic components at different granularity levels, in which self-awareness, semantic self-organization, self-similarity, and autonomic component aware principles can converge. These projects often assume the existence of an underlying infrastructure which can collect data from physical devices and sensors. Sometimes the design and the implementation of this low-level infrastructure are not considered at all since they are out of project scopes. In fact, the use of device simulators is a valid approach to testing high-level software components, postponing the issues related to data gathering.

However, the existence of an infrastructure allowing a systematic data collection process is fundamental both to implementing working prototypes of autonomic components (and other high-level abstractions) and for engineering the development of pervasive applications. The main goal of this paper is to introduce the requirements and the issues related to the design and the implementation of a solid substratum, able to provide sampling and first data processing functionalities, abstracting from devices' specificities. We will show how this goal can be achieved, both through the definition of a query language and the implementation of a middleware infrastructure.

Many different approaches have been designed to interact with the physical devices composing a pervasive system. The most common interfaces are procedural languages, virtual machines, message-oriented middlewares, etc. In this paper, a database-like abstraction of the pervasive system is proposed since the similarities among the "world of data" and the "world of sensors" suggest that it could be a good approach. In the "world of data," the abstraction concept has been widely used [2]. The main issue when managing information is that the same data can be stored on different storage platforms. Databases solve this issue by providing a high-level homogeneous view of data, regardless of their physical representation details. In the "world of sensors," a similar problem exists since many different technologies and communication protocols have to be managed in order to retrieve the needed information. Whereas each sensor node can be thought as a source of raw data, the goal of a pervasive system middleware is similar to the goal of a *Database Management System* (DBMS): It should provide a high-level homogeneous view of the heterogeneous devices, masking how data are generated. Moreover, DBMS research identified a standard query interface (the *SQL* language) to manipulate data and retrieve final results. A similar approach has been used in *Wireless Sensor Networks* (WSNs); in fact, some declarative languages have been defined to control the behavior of a sensor network. A goal of the PerLa project is to extend this approach in order to support a whole pervasive system. However, the database abstraction in the "world of sensors" should handle some additional issues that do not exist in traditional databases. The dynamics of the nodes must certainly be considered since a device can frequently join or leave the pervasive system, for example, when the status of the battery or of the wireless connection changes. Especially for monitoring purposes, the use of a declarative language is the approach that better allows to abstract the pervasive system, hiding the technical details related to device access. The advantage of defining a *SQL-like* language is the reduction of the effort needed by a user already experienced with standard SQL to learn it.

Context Awareness is another important feature of Pervasive Systems [3]. Most of the variables defining a context can be given values gathered from environmental sensors [4]; therefore, even if in its present development status PerLa does not envisage explicit context management functions, we are actively working toward extending it in order to manage dynamic contexts as well (see Section 6).

The rest of this paper is organized as follows: In Section 2, the most relevant projects, dealing with the different aspects of pervasive and ubiquitous systems, are introduced and compared with the goals and functionalities of PerLa.

Section 3 presents the description of the *PerLa* system together with the main reasons for the introduction of the new architecture composed of the *PerLa Language* [5] and the *PerLa Middleware*: The first one is a SQL-like language that allows one to define how data have to be gathered from the pervasive system and how collected data have to be processed; the second is the system development and language runtime support.

Section 4 describes two case studies in which PerLa has been employed: a wine production process monitoring system and a rockfall monitoring system.

Section 5 presents an evaluation of the system by analyzing the middleware performances and by highlighting the development speed improvement achieved through the use of PerLa.

Finally in Section 6, some concluding remarks and future work are presented.

2 State of the Art and Related Works

The term "pervasive computing" refers to a wide set of applications, and it encompasses many research areas, from networking and distributed computing to software engineering, knowledge management, and databases. Many are the existing projects, both in the academic and the industrial world, that could be considered part of the pervasive computing research or strictly related to it. A wide and complete survey on all these works is certainly interesting, but it is out of the scope of this paper. A selection of the most representative projects is, instead, useful to outline the most considered research aspects and to find out the less studied ones. Thus, the introduction of a nontrivial definition of pervasive system allows us to operate this selection, excluding all the projects not directly related to the research field.

The idea of ubiquitous computing, which is the key concept pervasive systems are based on, was first introduced by Weiser in [6]. He thought of ubiquitous computing as a

human-machine interaction model, characterized by the replacement of existing desktop PCs with a widespread diffusion of hardware and software components into everyday life objects. In this way, technology *becomes almost hidden to the final user* and it plays a background role. In fact, it is distributed on a widespread network of heterogeneous devices, called a pervasive system.

Afterward, Kindberg and Fox [7] pointed out two principles a system should respect in order to be considered pervasive, in accordance with the definition given by Weiser. These two characteristics are *physical integration* and *spontaneous interoperability*. The first one refers to the ability of hardware and software components to integrate and hide themselves into everyday life objects, interacting with the environment in a transparent way; the second refers to the devices' abilities to start communicating whenever required by the context, without explicit developer scheduling.

Existing projects related to pervasive systems seldom cover both the aspects defined by Kindberg and Fox: They mainly focus on a single feature. The works that try to reach a high level of integration typically deal with low-level hardware and software, and they are rarely interested in obtaining spontaneous interoperability features. Vice versa, projects that concentrate their efforts on spontaneous interoperability usually consider the integration issues only at a high abstraction level. They are not interested in the details allowing device integration, and they require the devices to expose high-level interfaces in order to communicate with the system (e.g., XML over HTTP protocols).

A complete pervasive application is composed of many architectural layers requiring different abilities, from the low-level hardware programming to the design of high-level abstractions. Physical integration is achieved by operating on low-level layers, while spontaneous interoperability is more related to the highest levels. This is one of the reasons that make it difficult and unusual to achieve both goals within the same research project.

Several papers available in the technical literature provide different surveys that aim at classifying existing projects related to pervasive systems; each of them proposes one or more dimensions a comparison can be based on. As an example, Messer et al. [8] cross the functionality layers (e.g., sensing, context management, execution platforms, user interaction, etc.) with the pervasive properties (e.g., device mobility, user mobility, software mobility, power management, adaptation, etc.). Salem and Nader's analysis [9] focuses instead on the programming model adopted to write the code deployed on the devices. Information about the compliance level with the expected pervasive system features is also provided for each considered project.

The analysis of the literature highlighted two criteria to cluster the projects related to pervasive systems: The first one is based on the covered architectural layers, while the second one is based on the middleware goals. Addressing the second criterion, there are monitoring projects that provide some interfaces to perform explicit monitoring and user-oriented projects that hide the technology and adapt their behavior with respect to the context. Basically, low-level projects are usually monitoring oriented, while high-level projects are usually user oriented.

Even if we do not want to deepen the topic here, a final remark is in order to relate pervasive systems architectures to other classes of systems which are already established or in course of development. An obvious thought goes to *Distributed Databases* [10], where, since the 1970s, a lot of work has been made on data allocation and query optimization. Much of this work is continuing on in the domain of *Cloud Computing* [11]. The same can be said about the convergence claim for ubiquitous computing technologies and the *Grid* [12]; while the Grid has been traditionally mainly considered a tool for High Performance Computing (HPC), quoting [13], "... it is now becoming a more generic platform for sharing any kind of networked resource and a Global Grid Forum Research Group Ubicomp-RG has been established to explore the synergies between the two types of systems." Moreover, the advent of wireless grids [14] makes the integration in a pervasive system of very heterogeneous devices, such as sensor-endowed cellular phones, concretely feasible.

In the following, a list of representative projects related to pervasive systems is considered, detailing the covered architectural layers and highlighting the reasons motivating the development of PerLa, the system presented in this paper. The first class of projects we cite is composed of all the monitoring projects ad hoc developed for specific applications. They are usually based on ad hoc designed boards, running low and high-level software components that are implemented from scratch and completely fitted on the given application. The routines for managing sensors, transmitting sampled data, and elaborating gathered data according to the application requirements are designed and implemented for the adopted board. Usually, these kinds of projects do not provide high-level devices abstractions, but rather they embed the system behavior customizing the code of each involved architectural layer. Some examples of these kinds of work can be found in [15], [16], [17], and [18]. There are many real applications that require the monitoring of some physical sensors in order to collect environmental data: Developing an ad hoc system for each of these applications requires low-level programming skills and makes very difficult both reusing the code and providing support for the addition and the integration of new devices. PerLa also aims at supporting this kind of monitoring application and speeding up the design and the development phases by providing a general middleware that allows one to easily integrate different kinds of devices and to reduce the amount of the needed low-level code.

The above-presented projects hard-code the behavior of each device, but they do not usually provide either a user interface or a network abstraction that allow one to modify the behavior of the whole system. *TinyDB* [19], [20] is the first project that introduced the idea of abstracting a Wireless Sensor Network (WSN) as a database. In this way, the final user can define the desired data gathering by writing a SQL-like query. At the higher level, an interface is provided to inject queries in the system. Each query is sent from the user workstation to the WSN base station; then, the query is distributed to all the nodes, and data are collected from the sensors in the environment, filtered, possibly aggregated, and routed out to the base station. TinyDB is defined over

TinyOS [21], and it exploits a power-efficient in-network processing algorithm. The main constraint of TinyDB is that only homogeneous WSNs composed of TinyOS-based devices can be managed. The idea behind PerLa is to extend the database-like abstraction from a single WSN to a whole pervasive system and to provide full support for heterogeneity, both at runtime and deployment time.

The *DSN* [22], [23] project is similar to TinyDB: The main difference between them is the interface the final user is provided with. In fact the DSN user can control the system through a declarative language, called Snlog that is a dialect of Datalog. Although this is an interesting approach, in PerLa we decided to adopt a SQL-like language since we think it could be easily learned by a final user already experienced with databases.

Both in the academic and in the industrial world, some projects have been developed trying to overcome the heterogeneity issues in order to allow the management of a high number of heterogeneous devices belonging to different technologies and exhibiting different features. These projects do not often provide an infrastructure to perform high-level processing of information, but only to collect gathered data in a centralized database. The definition of sampling details and of device access interfaces are clearly the goals of these works.

SWORD [24] is an example of industrial project developed by Siemens; it provides support for remote control in machine-to-machine scopes, and it allows a central management of a distributed infrastructure. SWORD is able to detect and signal alarms coming from the nodes deployed in the monitored area. All the detected signals are then collected in a centralized database that is the data source for the final control and monitoring application. The interaction between SWORD and the nodes is performed through a communication protocol based on the interchange of XML messages over an HTTP channel: This means that each device must support this protocol in order to be integrated with SWORD.

The *GSN* [25], [26] middleware is completely developed in Java, and it is executed on the computers composing the backbone of the acquisition network. The interface between the middleware and the devices is defined by *wrappers*: They are software components able to transform raw data coming from a physical source, to a GSN suitable format. GSN middleware is based on the concept of *Virtual Sensor*, that is a software component able to filter and process data coming from the wrappers or from other virtual sensors. The behavior of these components is defined through an XML configuration file, while the required data processing is set using a SQL-like query. Data coming from wrappers are processed through a chain of virtual sensors in order to produce output information. Unfortunately, Virtual Sensors have to be developed specifically for every new kind of node.

A remarkable feature of both [24] and [25] is the support for heterogeneous devices which is achieved by posing strong constraints on the supported nodes. In fact, support for Java programming at the device level is assumed or, at least, the availability of a TCP/IP stack is required; this requirement cannot be feasible on limited or ad hoc devices.

An interesting project is *TinyRest* [27], which exploits REST [28] principles, applying them to MOTE devices; more specifically, it defines a mapping between HTTP REST messages and TinyOS messages in order to allow a high-level sampling control. The approach of TinyRest does not require the device to implement a full TCP/IP stack, but rather a software layer is provided in order to enable HTTP data exchanges between the middleware and the devices. The approach is bounded to MOTE technology: PerLa aims at extending and improving it in order to support heterogeneity without posing many constraints on the devices.

In fact, the heterogeneous network support layer provided by PerLa tries to adapt the middleware itself to nodes' features rather than forcing the devices to support complex high-level protocols, allowing in this way a reduction of the coding effort required to a developer in order to integrate new devices. This makes the communication subsystem of PerLa quite different from both existing message-oriented technologies (e.g., *JMS* [29]) and tightly coupled communication middleware (e.g., *RMI* [30], *CORBA* [31]), that all require the presence of a Java Virtual Machine (JVM) on each device.

All the projects presented up to now aim at handling the sampling operations and managing heterogeneity issues. However, there are some projects that consider support to pervasiveness in a different light: They focus on the elaboration of gathered data, and they provide system ubiquity features through the concept of context.

The main research projects belonging to this group are *AURA* [32] and *GAIA* [33]: They both provide middleware architectures supporting complex devices, such as digital cameras, PDAs, shared storage spaces, overhead projectors, etc. One of their goals is to integrate all these kinds of devices, but the existence of device drivers (that provide high-level APIs suitable for interacting with each node) is assumed.

AURA introduces the concept of *personal aura*, a proxy that enables the user to use his own resources independently of the physical location the system is accessed from. In this way, pervasiveness is granted since the user can operate independently of his location and of the device used to interact with the system. GAIA introduces some ideas that are very close to AURA ones: When the user moves from a physical environment to another, available technologies detect his presence and tune the system to better support his activities. GAIA reaches this goal by hiding the technologies in the environment (video motion, speech recognizer, etc.), achieving in this way a high ubiquity level.

The *CoolTown* project [34] is quite similar to AURA and GAIA, but it was developed in the industrial world at HP Labs. Its goal was the implementation of an ubiquitous system able to adapt its behavior according to the specific context. The system is completely web-based, and each device is abstracted through a web interface that allows system-node interaction.

PerLa is designed to cover the lowest architectural layers of a pervasive system and to act as a middleware platform over which application logic can be developed without taking care of device integration issues. These features make PerLa a powerful tool to manage monitoring applications that are usually handled by systems like GSN, SWORD, TinyDB, or DSN. Although PerLa also supports the management of contexts, it is not designed to

replace systems like AURA, GAIA, and CoolTown, but rather to provide a solid substratum over which this kind of application can be deployed. In fact, the user-oriented features provided by these projects are out of the scope of the PerLa project; vice versa, the technical details to interact with physical devices are out of the scope of AURA, GAIA, and CoolTown.

The maximum degree of abstraction is achieved by *CASCADAS* [35], which aims at designing autonomic components, able to self-organize and to adapt to the situation in order to achieve the expected functionalities. The goal of the project is the development of the knowledge and the techniques needed to build an autonomous computing network. The resulting system should properly work without requiring continuous administrator supervision; moreover, it should be able to learn its own optimal behavior over time and to integrate new components when needed. The CASCADAS project does not take care of low-level issues related to network architecture and sampling operations. On the contrary, the existence of a high-level network service, able to produce data streams and feed autonomic components, is assumed.

The *SelfLet* project [36], [37] that is under development at Politecnico di Milano is part of CASCADAS. It also deals with autonomic software components, but it mainly focuses on software engineering aspects, such as the life cycle description and the definition of runtime validation models. For both these aspects, the SelfLet approach highlights how the design phase and the deployment phase can no longer be considered as two completely distinct activities. The project development is bounded to high-level layers. A Java-based implementation, currently in embryonic phase, relies on several existing technologies and tools (REDS [38], DROOLS [39], LIME [40], OSGi [41]) for the development of SelfLet internal components. Therefore, each SelfLet can be thought as a Java object, able to achieve a goal through the execution of a *Behavior*. This behavior can be accomplished by searching for services that are provided by other SelfLets or executing an elementary activity, called *Ability*. Research interests are focused over the ability level. The routines performing sampling, actuation, and communication with physical devices are not part of the SelfLet middleware, and they must be implemented as abilities.

The *MoGATU* project [42], [43], [44] aims at extending the functionalities of mobile devices by exploiting their ability to autonomously establish connections among them. The main difference with respect to the other projects cited consists of the different role reserved for mobile devices. In fact, they are not treated as the sensors and the actuators needed to solve user submitted queries, but rather they are intended as the interfaces used by the end users to access some services offered by the system.

This project employs powerful devices equipped with a user interface, like PDAs and laptops, where client applications act as data consumers and give ubiquitous access to the information stored in the wired network. These devices also operate as data producers, providing neighbor nodes with information retrieved from wired servers or other nodes.

Prism-MW [45] is an extensible middleware platform designed to enable efficient implementation, deployment, and execution of distributed software systems. In Prism-MW, the heterogeneity support is entirely focused on enabling communications among different devices equipped with different operating systems and programmed with different languages. PerLa also extends the heterogeneity support to sampling operations and data management. The main goal of both projects is to provide a highly decoupled component-based extensible framework to develop distributed applications; in Section 3, we shall further compare their architectures.

Glide [12] is a Java platform developed on the top of Prism-MW. The goal of the system is to allow the search and the retrieval of resources (e.g., MP3 files) hosted on a set of nodes and characterized by some attributes (e.g., genre, quality, etc.), given a query in terms of these attributes (e.g., retrieve all the MP3 files with genre equals pop and quality equals high). GLIDE and PerLa are very different systems since PerLa focuses on monitoring applications and it is designed to support sampling operations and to manage data streams while GLIDE is designed to search and manage single resources. However, some similarities can be found between GLIDE and the Registry component of PerLa, as we shall see in Section 3.

3 The PerLa System

As highlighted in the earlier sections of this paper, there are currently no projects that provide a comprehensive and adequate solution for managing a modern pervasive system. The deficiencies identified in our analysis of the state of the art always result in poor usability for the end users and/or inadequate support for node developers and pose great limits to the scenarios of application where these middlewares can be successfully employed. PerLa aims at overcoming most of the weaknesses of existing management systems for sensing networks by achieving a coverage of most of the layers identified in Section 2.

The development of the PerLa System focused on the design and implementation of the following features:

- data-centric view of the pervasive systems,
- homogeneous high-level interface to heterogeneous devices,
- support for highly dynamic networks (e.g., wireless sensor networks),
- minimal coding effort for new device addition.

The first two targets are achieved using a technique already known in the literature (see [19], [20], and [23]): managing the pervasive system as a database. The result of this approach is the possibility of accessing all data generated by the sensing network via an SQL-like query language, called the PerLa Language, that allows end users and high-level applications to gather and process information without any knowledge of the underlying pervasive system. Every detail needed to interact with the network nodes, such as hardware and software idiosyncrasies, communication paradigms and protocols, computational capabilities, etc., is completely masked by the PerLa Middleware.

The PerLa Middleware provides great scalability both in terms of *number* of nodes and *types* of nodes. Other middlewares for pervasive systems only support deployment-time network configuration (see [24], [25], and [26]), or

TABLE 1
A Comparison of Middlewares for Pervasive Systems

	AURA	GAIA	MoGATU	TinyDB	SWORD	GSN	DSN	Cascadas	Selflet	PRISM	GLIDE	PerLa
Support for autonomic components								✓	✓			
Support for contexts	✓	✓	✓					✓	✓			✓
Data manipulation	✓	✓	✓		✓	✓		✓	✓		✓	✓
Centralized node integration	✓	✓	✓		✓	✓		✓	✓		✓	✓
Run-time heterogeneous network support												✓
Deployment-time heterogeneous network support					✓	✓				✓	✓	✓
Low level support				✓			✓					✓

provide runtime device addition capabilities for a well-defined class of sensing nodes at best (e.g., TinyDB [19], [20]).

These limitations are no longer acceptable. In modern pervasive systems, specifically wireless sensor networks, nodes can hardly be considered "static" entities. Hardware and software failures, device mobility or communication problems can significantly impact the stability of a sensing network. Furthermore, the ever-increasing presence of transient devices like PDAs, smart-phones, personal biometric sensors, and mobile environmental monitoring appliances makes the resilience to network changes an essential feature for a modern pervasive system middleware. Support for runtime network reconfigurability is therefore a necessity. Moreover, the middleware should also be able to detect device abilities and delegate to them whichever computation they can perform. The tasks that nodes are unable to perform should be executed by the middleware.

PerLa fulfills these requirements by means of a *Plug & Play* device addition mechanism. New types of nodes are registered in the system using an *XML Device Descriptor*, i.e., a document containing all the information needed to interact with the hardware and software modules of the sensing nodes. The PerLa Middleware, upon reception of a device descriptor, autonomously assembles every software component needed to handle the corresponding sensor node. End users and node developers are not required to write any additional line of code. State-of-the-art middlewares for sensing networks, like [24], [25], [42], [33], require instead the creation of individual software wrappers for every device connected to the system.

Table 1 provides a feature-wise comparison among PerLa and the other systems analyzed in the state-of-the-art section of this paper.

3.1 PerLa System Overview

The PerLa System is composed of different software modules with well-defined interfaces and functions.

To provide a practical and effective framework that would support the implementation of middleware components and ensure a strong decoupling among them, the first stages of the PerLa architectural design were spent developing a set of Java foundation software elements. The goal of these basic building blocks is to provide the common features needed by any PerLa component, such as initialization, event logging, and communication functionalities. The following list contains a brief digest of the aforementioned elements:

- *Component*. The base abstract class inherited by every middleware block. Defines the essential methods employed to manage all PerLa subsystems (e.g., start, stop, etc.).
- *Pipe*. This object is a one-way message queue that provides a highly decoupled communication system between different middleware components. Transmission of information through pipes is performed using the primitives *push* and *pop*, which provide a mechanism to, respectively, send and receive messages. Each PerLa component can be connected to one or more input and output pipes.
- *Waiter*. This class allows synchronized waiting on multiple pipes.
- *Channel*. The Channel is an abstract class that acts as a gateway between a pipe and a physical channel.

Not surprisingly, a similar architecture had been already studied and described in the literature [45]. Different analogies can be made among the two designs; the Component class of Prism-MW core is very similar to the Component class in PerLa: The *Send()* method is the equivalent of pushing a message into an output pipe, while the *Handle()* method is similar to the reception of a message from a pipe using a Waiter object.

As shown in Fig. 1, PerLa components are divided into two macrolayers:

- *Low-level support*. Low-level support provides a homogeneous API to manage and operate all the different devices that compose the pervasive system.
- *High-level support*. This layer implements data management and query execution services needed to interface end users and applications with the pervasive system.

The end users and the application developers interface with the PerLa system through the upper interface with the High-Level Support, while the "network architects" (e.g., node developers), who work near to the physical devices, interface with the system through the Physical Layer interface (Fig. 1).

The main software module of the low-level support layer is the *Functionality Proxy Component* (FPC). As suggested by its name, the FPC's primary task is to act as a proxy among sensing nodes and the rest of the PerLa System. Due to the FPC's homogeneous access interface, components of the High-level support layer can communicate with the

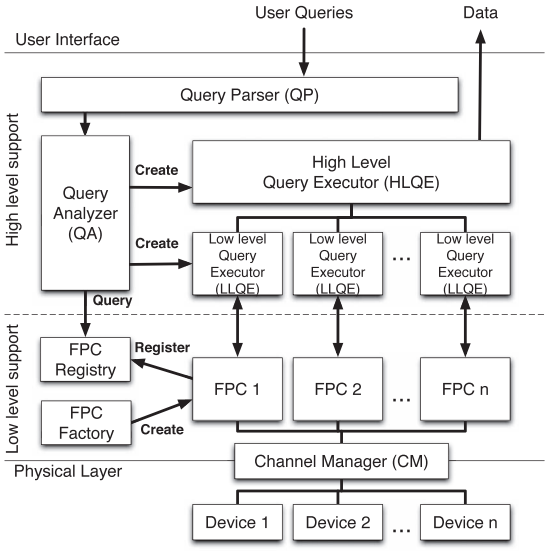

Fig. 1. PerLa middleware architecture.

physical devices, ignoring their functional behavior. As a matter of fact, due to the abstraction provided by the Functionality Proxy Component, all nodes of the pervasive system appear to implement a common interface.

The physical communication between sensing devices and FPCs is completely managed by the *Channel Manager*. This component handles the creation of *Virtual Channels*, a software abstraction that hides all the peculiarities of the medium adopted to communicate with the pervasive system nodes. By means of virtual channels, FPCs and physical devices can communicate transparently.

PerLa does not impose any constraint on the communication protocol and message format that the sensing nodes must implement. In fact, PerLa allows the use of arbitrary protocols. Messages to or from the nodes are automatically converted by the FPC; this feature significantly reduces the effort in terms of node firmware development and does not require a specific physical communication channel. PerLa can, therefore, operate with a wide range of devices, regardless of their hardware and software capabilities. Other approaches (e.g., [24], [33]) impose simple and clear high-level middleware interfaces that are difficult to implement on certain kinds of sensing nodes (e.g., an HTTP interface can be too hard to implement if the physical device is a μ-controller connected to the middleware via an industrial bus).

PerLa is not the only middleware to provide this type of support for device heterogeneity. As noticed in Section 2, a similar technique is presented in GSN [26], [25]. However, in this case, the integration of a new kind of device requires node developers to code the corresponding Wrapper. Heterogeneity support is therefore limited to *deployment-time*.

Conversely, PerLa supports the introduction of new kinds of nodes at *runtime* through a *Plug & Play* device addition mechanism. FPCs are created by means of the *FPC Factory*, a software module responsible for PerLa's Plug & Play system. New nodes can join a network managed by PerLa simply by forwarding their XML device descriptor to the FPC Factory, which creates a Functionality Proxy Component suitable to handling the node. Further details will be introduced in Section 3.1.2.

The complete inventory of all FPCs active in a PerLa installation is maintained by the *FPC Registry*. This component is mainly employed to select the list of devices that can run a particular query (see also Section 3.2.3), and to check if a node trying to connect with the PerLa System already has a corresponding FPC (mobile and transient device support). The *Query Parser* and the *Query Analyzer* compose the user interface of the entire PerLa System. The PerLa Language defines two classes of queries:

- *Low-Level Queries* (LLQ). Define the behavior of single devices.
- *High-Level Queries* (HLQ). Manipulate data streams generated by LLQs.

A detailed explanation of the roles and the differences between the two query levels is discussed in Section 3.2.

The Query Parser is PerLa's front end to final users. This component receives and parses textual queries and sends them to the Query Analyzer. As shown in Fig. 1, the Query Analyzer is responsible for the creation of both the *High-Level Query Executor* (HLQE) and the *Low-Level Query Executor* (LLQE).

The High-Level Query Executor is implemented as a wrapper for an external *Data Stream Management System* (DSMS)[46]. Its main purpose is to allow users to perform data management operations on the information gathered from the pervasive system. Each High-Level Query submitted by the end user is executed on a dedicated HLQE.

Differently from High-Level Queries, Low-Level Queries required the creation of an ad hoc execution engine, called the Low-Level Query Executor. The execution of a single-Low-Level Query is performed according to this procedure:

- the Query Analyzer accesses the FPC Registry to retrieve a list of FPCs that can successfully perform the query (see Section 3.2.3, *EXECUTE IF* clause);
- the Query Analyzer instantiates a Low-Level Query Executor for each FPC in the list and configures it for the execution of the statement submitted by the user;
- each Low-Level Query Executor interacts with the corresponding FPC to gather information from the physical devices.

All records generated by the LLQEs related to a single query are then combined into a single data stream, which represents the output of the Low-Level Query sent by the user.

3.1.1 Functionality Proxy Component

An FPC is a Java object that provides a functional abstraction of a given device or a group of them. The FPC is used by other PerLa middleware components to interact with the sensing node itself. A hardware device can host an FPC only if:

- it runs a *Java Virtual Machine*,
- it is connected to the PerLa Middleware via *TCP/IP*.

Although some physical devices are powerful enough to host their own FPCs, many others are not, either because

they cannot run a JVM or because they are not provided with a TCP/IP interface.

In the deployment presented in Section 4.1, all FPCs are hosted in the central network node, which is powerful enough to support a complete JVM. Conversely, due to strict hardware limitations on both the nodes and the gateway, the deployment shown in Section 4.2 requires all FPCs to be run on a remote server. In any case, the location of the FPC should be as near as possible to the abstracted device.

The FPC actually implements a *Logical Object*, i.e., an abstraction of the physical node that is used by the Low-Level Query Executor to access the information gathered from the sensing nodes. By virtue of this abstraction, the PerLa Language is completely independent of any hardware or software feature exhibited by the physical devices (see Section 3.2.3 for further details).

The information generated by the physical sensing nodes are abstracted as FPC *Attributes*. Attributes can be used to retrieve the node state (e.g., battery status, memory occupation, etc.), to access sampled data (e.g., temperature, pressure, etc.), or to change some parameters on the device (sampling frequency, node parameters, etc.). FPC Attributes can be classified in three categories, depending on the type of value they abstract:

- *Static attributes.* Represent constant values that describe immutable characteristics of the sensing node. They are commonly used to define properties like device name, maximum sampling rate, location of stationary nodes, etc. Attributes of this type are set using the XML device descriptor, and do not change their value for the entire FPC life cycle.
- *Probing dynamic attributes.* When this kind of attribute is read, the FPC must refer to the physical device in order to produce the requested value. Similarly, when a write operation is performed (if supported), the FPC has to interact with the device to set the new attribute value. Probing attributes can be used to represent a real sensor (e.g., temperature, pressure, etc.), a real actuator (e.g., the position of a stepper motor), or a memory area (e.g., information stored in a device RAM, ROM, flash, etc.). Read and write operations on a probing dynamic attribute always result in an interaction with a physical node. Should the node be momentarily unavailable, the FPC caches and defers the operation to a more convenient time (i.e., writes and reads are performed when the node is back online or when the network connection allows to communicate with the device). Even if cached, every single user operation on an FPC's probing dynamic attribute is relayed to the devices.
- *Nonprobing dynamic attributes.* When a nonprobing dynamic attribute is read, the FPC returns the value stored in its local cache. No actual reading is performed on the physical device. The cached value is periodically refreshed in accordance with the guidelines set in the XML device descriptor. Two policies are available: pull (the FPC periodically fetches the new value from the device, regardless of user requests for that attribute) and push (the device sends a message to the FPC whenever a new value is available). Writing operations are not supported on nonprobing dynamic attributes by design since the semantics associated with them would imply the possibility of losing messages if not sent to the node before another write is requested by the user. This behavior is not desirable when, for example, a series of command are to be written on the device, since the PerLa system would not be able to guarantee the delivery of all of them.

It is important to note that all three types of FPC attributes are accessed using the same paradigm, and that different access semantics can be added in the future if necessary.

Events and notifications fired from a physical device (e.g., when an RFID reader senses an RFID tag or when a device identifies a certain pattern in the signal produced by a sensor) are intercepted by the FPC and relayed to the other middleware components. This feature allows PerLa to support both *push* and *pull* data retrieving semantics. Push nodes (like those used in Sections 4.2 and 4.1) spontaneously send the data messages to PerLa. Conversely, pull nodes send information only on request. The FPC is therefore required to periodically query the device if more than one sample is needed.

As said before, an FPC can be defined to abstract either a single node or a group of them: In the latter option, the final user controls all the nodes as a unique entity. The machine hosting an FPC that controls multiple devices must be able to sustain the workload required to manage numerous data sources.

Table 2 shows a real example of the mechanism that allows to map a node message and the Java object used by FPC. In particular, Table 2a presents the message sent by a node (used in the deployment in Section 4.1), which consists of five attributes (i.e., `timestamp`, `panelVoltage`, etc.) with different byte length and sign. Moreover, the μ-controller of this node adopts "big endian" encoding. In Table 2c, the corresponding Java class is presented. The information about fields length, sign, and endianness are stored in special Java annotation (setter getter methods and constructors are omitted in the example). By means of these annotations, the FPC can unmarshal the byte-stream sent by the device into the corresponding Java class.

More complex data structures (e.g., used to store microseismic events presented in Section 4.1) are unmarshaled in a hierarchical fashion. The resulting class reflects the nested C-structure used in the node message.

Fig. 2 provides an overview of the Functionality Proxy Component, and it represents the interactions between this component and the device. It also distinguishes the software modules belonging to the middleware from the software modules written by the developer who is integrating the device with the PerLa middleware.

The upper interface of an FPC is composed of two classes of methods: The first one (`getAttributeByName()`) allows one to retrieve the data coming from the device, while the second one (`setAttributeByName()`) allows one to set parameters on the device. In the lower interface, it is shown how the node sends messages (of the previous example) and the FPC (in particular the unmarshaler) decodes and stores them into an internal object (`DataMessage` class). However,

TABLE 2
Physical to Logical Attributes Mapping

C-like message structure

```
typedef struct{
  uint64_t timestamp;
  int16_t panelVoltage;
  int16_t panelCurrent;
  int16_t batteryVoltage;
  uint8_t flags;
} DataMessage;
```

a)

XML description

```
<parameterStructure name="DataMessage">
  <endianess>BigEndian</endianess>
  <parameterElement name="timestamp">
    <length>8</length>
    <type nameType="long">
      <sign>unsigned</sign>
    </type>
    <attributeType>probing</attributeType>
    <permission>r</permission>
    <continuousValue />
  </parameterElement>

  <parameterElement name="panelVoltage">
    <length>2</length>
    <type nameType="long">
      <sign>signed</sign>
    </type>
    <attributeType>probing</attributeType>
    <permission>r</permission>
    <continuousValue />
  </parameterElement>
  ...
</parameterStructure>
```

b)

Java Class

```
@StructInfo(endianness = Endianness.BIG_ENDIAN)
public class DataMessage{
  ...
  @SimpleField(size = 8, sign = Sign.UNSIGNED)
  private long timestamp;

  @SimpleField(size = 2, sign = Sign.SIGNED)
  private int panelVoltage;

  @SimpleField(size = 2, sign = Sign.SIGNED)
  private int panelCurrent;

  @SimpleField(size = 2, sign = Sign.SIGNED)
  private int batteryVoltage;

  @SimpleField(size = 1, sign = Sign.UNSIGNED)
  private int flags;
  ...
```

c)

this class is created at runtime; hence, a suitable way to access its field consists of the use of the Java Reflection mechanism. The proposed solution is general and does not depend on the given class (e.g., `DataMessage` of the example).

In the same way, `setAttributeByName()` allows the device parameter change by manipulating the corresponding class (e.g., `ParamMsg`). The marshaller uses this class to create a message for the node; in turn the node sets the corresponding parameters after receiving the message.

3.1.2 FPC Factory and Registry

As presented above, an FPC allows a simple interaction with a physical device, hiding the communication details and masking low-level programming issues. To reach a real Plug & Play behavior, the middleware should be able to create FPCs at runtime, to instantiate them on a Java and TCP/IP enabled machine, and to set up a Virtual Channel (see Section 3.1.3). A middleware component, called FPC Factory, has been designed to achieve this goal. An instance of this object is deployed on each machine charged to host the FPCs; given the information contained in an XML device descriptor file, the FPC code is automatically generated. This process is feasible because the FPC can be implemented by merging a certain number of modules, each of them managing a different aspect of the interaction with the physical node. The XML descriptor is used to decide which are the right modules to be selected from a library in order to provide a proper device abstraction. To provide full Plug & Play support, a physical device that is joining the system must be able to send its XML descriptor to the nearest factory and to start communicating with the generated FPC without any human interaction. A special Virtual Channel is reserved on each physical channel to communicate with the FPC Factory. Thus, the new device sends its XML descriptor on the reserved channel and, if no error occurs, the binding of the generated FPC with a new Virtual Channel is notified. All the subsequent communications between the physical device and the FPC are performed on the new Virtual Channel.

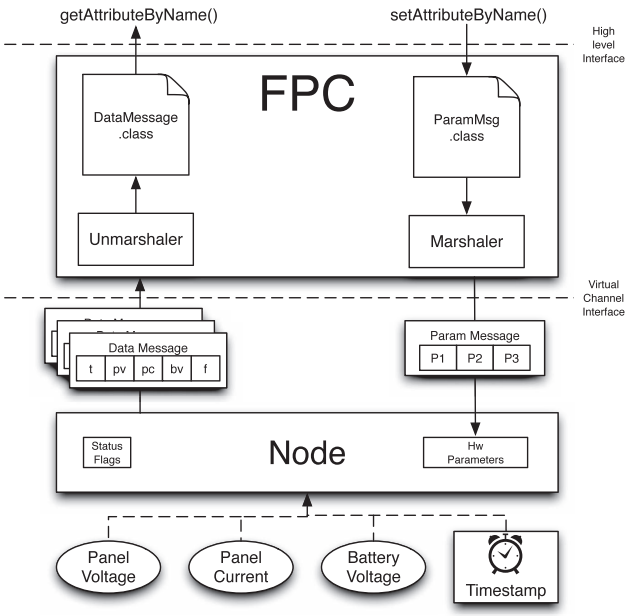

Fig. 2. Functionality proxy component.

All the relevant information needed to dynamically build the FPC and to establish a data connection between the physical device and the generated FPC must be included in the file. More specifically, the developer has to specify the details needed to contact the available FPC Factories during the setup phase. They include the initialization parameters of the physical channels required to establish a communication with the FPC Factory, the list of available sensors, and the physical measures that can be sampled by the device. For each attribute, the name, the data type at language level, the data type at physical level (e.g., the encoding and the length of the value generated by the *Analog to Digital Converter* (ADC)), the conversion function, and the constraints on the sampling rate should be specified.

A working example of an XML device descriptor is given in Table 2b. As mentioned before, the FPC (by using the generated class presented in Table 2c) deserializes the encoded messages sent from the node (see Table 2a).

Another important middleware component strictly related to the FPC Factory is the FPC Registry; it is basically a Main Memory Database (MMDB) in which a reference to any existing FPC is maintained. Its main goal is to provide support for the evaluation of the EXECUTE IF clause of Low-Level Queries. After a new FPC is generated, the factory registers it; the device metadata (supported attributes and events, maximum sampling frequencies, etc.), the attribute values, and the remote reference to the FPC object are stored in the FPC Registry.

Assuming FPCs as the resources and properly translating the *EXECUTE IF* clause of PerLa queries (see Section 3.2.2) into GLIDE queries, the *Registry* can probably be thought as a set of *ProfileServer* and *ProductServer* components [12].

We can summarize the steps performed by the Factory to create a FPC:

1. the XML description is received by the Factory;
2. the Factory using the JAXB [47] library transforms the XML entity into a Java object (within annotations, constructors, setter, and getter methods);
3. the Factory creates a new FPC derived class, encapsulating the object generated in step 2;
4. the Factory invokes the Java compiler and produces the `.class` files;
5. the Factory invokes the Java loader to load the FPC at runtime;
6. the Factory registers the new FPC in the registry;
7. the Factory binds the FPC and the physical device through the virtual channel.

In a real pervasive system, a physical device is often connected to different nodes that can host its FPC, and more than one physical channel could exist between these nodes and the device. Moreover, some of the Java-enabled nodes and some of the physical channels can be temporarily unavailable due to power management policies or mobility problems. Thus, a mechanism to allow an already initialized device to reconnect to the middleware using a different physical channel has been implemented. Basically, the device executes the initialization protocol again, but the FPC Factory, simply by performing a search in the registry, recognizes that an FPC for that device already exists in the middleware. To restore the communication between the device and its FPC, the found FPC is reinstated and a new Virtual Channel is established.

3.1.3 Channel Manager

Network heterogeneity is an intrinsic characteristic of pervasive systems. Many different physical medium and protocols are often required, even in small deployments, to provide a communication backbone for the different technologies employed in a sensing network.

The PerLa Middleware deals with this problem using a specific component, the Channel Manager. This software module is responsible for the creation of Virtual Channels. Virtual channels use the existing communication facilities of the pervasive system to provide a bidirectional, point-to-point logical communication layer. By means of this software abstraction, the FPC and the corresponding sensing device can communicate among them as if they were connected through a dedicated link. The Channel Manager handles every routing and multiplexing operation required to conceal the physical communication difficulties (e.g., a single physical link shared among many sensing nodes).

Consider, as an example (given in Section 4.2), a radio channel that operates in duty cycle mode to reduce power consumption; in this case, the Channel Manager must support buffering in order to mask the real behavior of the physical link. Consider, as a second example (given in Section 4.1), a device that is connected to the nearest Java machine through two cascaded links (for instance, a slave device can be attached to a master *CAN-bus* [48] board, which, in turn, communicates with a Java machine through a radio link). In this case, the channel manager should also take into account the routing required to transport the messages from a Virtual Channel side to the other one; more generally, the channel manager must deal with heterogeneous physical networks.

Note that the same physical link can be shared by many couples FPC-device: Consider as an example a situation in which a certain number of nodes are cabled on a shared broadcast bus (e.g., CAN-bus) that must be used to communicate with the nearest Java machine. Thus, the channel virtualization module must also perform multiplexing on the physical channel in order to manage similar situations. Another important feature of the Channel Manager is the ability of automatically setting up the virtual channel when a new device joins the system: Only some configuration files are required to be installed on the physical device in order to inform the middleware code about the supported protocols.

The Channel Manager is implemented using two different programming languages: Java and C. While the Java version is deployed on the FPC side, the portable C version has to be executed on the physical nodes. The latter variant of the Channel Manager allows a great reduction in terms of code that needs to be written by node developers since all the libraries required to handle PerLa's Virtual Channels are distributed along with the PerLa middleware. Both Java and C implementations of the channel manager now support the most used bus interfaces (i.e., serial bus, TCP/IP sockets, CAN-bus), but a developer can integrate the middleware with new modules in order to support other protocols.

3.2 PerLa Language Description

As previously said, the PerLa Language aims at providing a database like abstraction of the whole pervasive system in order to hide the high complexity of low-level programming and allow users to retrieve data from the system in a fast and easy way. The main issue we dealt with during the language design phase was the definition of a small set of clauses having a large semantic expressiveness. In fact, our main goal was to obtain a language marked by an easy syntax (in order to be easily learned), but able to manage many kinds of heterogeneous devices and to support most of the different existing sampling modes (periodic, on event, etc.). Another goal of the language was to avoid the introduction of a specific clause for each nonfunctional feature, but rather to

provide a uniform mechanism to access both device metadata (sampling frequency, battery status, etc.) and the result of a sampling operation. This generic approach allows maintaining the language simplicity and simplifying the introduction of new nonfunctional characteristics that can be considered now or that will possibly emerge in the future. In this section, the PerLa Language is briefly introduced. First, existing sampling modes are explained; then, the types of supported statements are reported; finally, Low-Level Queries are presented in detail. An in-depth analysis and the full *EBNF* formal definition of the PerLa Language can be found in [5] and [49].

3.2.1 Physical Devices and Sampling Operation

The sampling operation has a key role in the definition of a language that should provide an easy interaction with a pervasive system. In fact, the main goal of a final user is to retrieve data from the network, and this can be done by querying a given subset of nodes and processing the data gathered from them. The wide variety of devices that can be part of a pervasive system suggests that many different kinds of sampling exist. Moreover, although the concept of sampling is extremely clear for many technologies, its definition is not trivial for other ones. Thus, identifying the most important sampling modes and defining a common abstraction is the main challenge in order to provide a simple and powerful language.

Many nodes that are usually employed in common monitoring systems are composed of an electronic board hosting one or more physical sensors, each of them able to probe a specific physical measure. In this case, a sampling operation is simply the reading of the sensor output (e.g., the value of an ADC connected to the signal conditioning stage of the sensor). If a node also has a storage device on board (e.g., RAM, ROM, flash, etc.), the sampling operation can be extended to the reading of a section of that memory. Note that, from the language point of view, there are no differences between a sampled value obtained from probing a sensor and a value read from memory. In fact, in both cases, the user requests a sampling and the device returns a value; in the following, we will refer to this behavior as *time-based sampling*.

Pervasive systems often deal with other kinds of devices for which the definition of the sampling operation is more difficult. Consider, as an example, the RFID technology: Two issues must be considered in order to provide clear semantics of the sampling operation. First, the subject and the object of the sampling must be identified. The RFID reader can be abstracted as the sensor, while the tag ID can be considered as the sampled datum. However, a dual approach is possible: Each tag can be abstracted as a sensor, while the RFID reader identifier can be considered as the sampled datum. The first approach is the most common one, but the second is more fitted to the requirements of a monitoring application. In fact, RFID tags are usually attached to the items the user wants to monitor. Thus, the user is often interested in discovering the sequence of readers "seen by a tag" rather than knowing the list of tags entering the range of a given reader. The second issue to be considered when dealing with RFIDs is inherently related to the features of this technology. In fact, a High-Frequency RFID tag is activated only when it crosses the limited operating range of a reader. When a sampling is required, there are high probabilities that no tag is active. There are two approaches to deal with this problem. The first one requires the introduction of a cached value: When a sampling is required, the returned value is the cached one, i.e., the ID of the last reader that sensed a given tag (or the ID of the last tag that was seen by a given reader). In this way, the sampling semantics is kept very similar to that of classical sensor sampling. On the contrary, the second approach requires a radical change in the sampling abstraction; in fact, the interaction model between the device and the upper software layers becomes asynchronous. The idea is to abstract the sampling operation as an event: When a tag is sensed by a reader, an event flag is raised and a parameter of this event reports the ID of the reader (or of the tag). Note that when the latter approach is used, the sampling operation is not invoked by the user; there is no concept of sampling rate, and the execution of the next sampling operation is logically delegated to the tag device. In the following, we will refer to this behavior as *event-based sampling*. Even if other sampling modes could be implemented, in the first implementation of PerLa we only support time and event-based sampling since these modes were required by the applications we have been involved in.

The remainder of this section introduces the different query typologies that compose the PerLa Language. The following language description is defined around the concept of Logical Object. As already explained in Section 3.1.1, the logical object is an abstract sensing or actuation device that exposes a common and uniform set of functionalities required to interact with the physical sensing node. The most important property of a logical object is the ability to expose a list of Attributes, i.e., the features that can be queried or modified on the physical device. Attributes are used to provide a high-level abstraction of a physical transducer, an actuator, or of a variable in the node's memory. In the next sections, the concepts of node, device and logical object are used interchangeably.

3.2.2 Supported PerLa Queries

After the analysis of pervasive systems peculiarities, we decided to support three kinds of statements in PerLa.

Low-Level Queries. Allow one to precisely define the behavior of a single device. The main role of a low-level statement is to define the sampling operations, but also to allow the application of SQL operators (such as: grouping, aggregation, filtering) on sampled data. When an LLQ is injected in the system, the set of Logical Objects that meet all the requirements to execute it is computed. Then, an instance of the query is deployed on each selected logical object; data produced by all these instances are collected in a stream that can be further manipulated by *High-Level Queries* or sent as output to the final user.

As said before, PerLa is completely declarative, and it is characterized by an SQL-like syntax. However, LLQs present some relevant syntactical differences with respect to standard SQL in order to provide the needed expressiveness for managing the interaction with logical objects (e.g., definition of the sampling parameters).

High-Level Queries. Allow one to define data manipulation operations on streams generated by Low-Level Queries or other High-Level Queries. These kinds of queries do not directly deal with the logical object abstraction since they operate only on streams independently from their sources. As a consequence no ad hoc clauses are needed: The syntax and the semantics of these statements are similar to those of streaming DSMSs. Detailed examples of High-Level Queries can be found in [49]

Actuation Queries (AQ). Add the expressiveness to modify the state of a device, enabling in this way a complete interaction with the environment. This kind of statement is often employed to control physical actuators or to set some parameters used by low-level algorithms directly executed on the node (e.g., thresholds to generate events). An example of Actuation Query can be found in Section 4.1. The complete semantics of this statement is detailed in [49].

3.2.3 Low-Level Queries in Detail

In this section, LLQs are explained in detail since they are the most interesting and original part of the language. Moreover, knowing the middleware design will be useful to understand some choices in semantics of these statements. After introducing LLQ statements from a syntactical point of view, two simple examples are reported in order to better clarify the language features.

Low-Level Queries are introduced by the clause *AS LOW:*, preceded by the structure of the stream in which the output records have to be inserted. Every LLQ is composed of four syntactical blocks, each of them defining a specific aspect of the query semantics.

The **Sampling Section** specifies how and when the sampling operation should be performed. The above-described logical object abstraction allows a formalization of this operation that is completely independent of the low-level details of the physical device: In fact, sampling is simply defined as the generation of a record whose fields are filled with the current values of the logical object attributes. As said in Section 3.2.1, event-based and time-based are the supported sampling types; the first one forces a sampling whenever an event is raised by the logical object, while the second one forces the sampling to be periodically executed with a certain frequency (that can be parametrically specified). To clarify the distinction between event-based and time-based sampling modes, consider the following example. Suppose we want to retrieve the list of RFID readers that sense a specific RFID tag. The logical object that wraps the RFID tag has at least one static attribute (the tag ID), one dynamic *nonprobing* attribute (the last RFID reader that sensed the tag), and one event (that signals the occurred sensing). In this situation, a reasonable sampling mode is the event based one: A sampled record is generated whenever the event is raised. The opposite situation is the case of a node having a temperature sensor on board that should be sampled every 10 minutes. In this case, the logical object that wraps the node has a dynamic *probing* attribute that returns the current temperature. To obtain the required sampling rate, a time-based sampling mode must be used. It should be noted that a logical object abstracting a device, characterized by an intrinsic type of sampling, can contain the logic needed to support other sampling types. As an example, a passive RFID tag is obviously inherently event based, but some meaningful queries with time-based sampling can be performed too (see the example in Table 3b).

TABLE 3
Low-Level Queries Examples

a)
```
CREATE OUTPUT STREAM Table (Temperature FLOAT)
AS LOW:
  EVERY 10 min
  SELECT MAX(temp, 10 min)
  SAMPLING
    EVERY 1 min
  EXECUTE IF EXISTS(temp) AND EXISTS(room)
    AND room = 3
```

b)
```
CREATE OUTPUT STREAM Table (rfid STRING, counter INTEGER)
AS LOW:
  EVERY 10 min
  SELECT lastReaderId, COUNT(*, 10 min)
  SAMPLING
    ON EVENT lastReaderChanged
  EXECUTE IF ID=[tag]
  TERMINATE AFTER 1 SELECTIONS
```

The **Data Management Section**, introduced by the *SELECT* clause, has the role of managing sampled data and computing query results. The interface between this section and the previous one is an ideally infinite buffer (in the following referred as the *Local Buffer*), where all the sampled records are appended. The data management section allows end users to specify which operations are to be performed on local buffer records to create the actual query output. The syntax of this query section strongly resembles standard SQL. However, important differences exist due to the need of managing the infinite data stream through the definition of record windows in the local buffer. Moreover, other differences have been introduced due to our decision of focusing especially on aggregates. In fact, we think that in many real situations the data of interest are an aggregation of some sampled values, rather than the list of all sampled records. This led to the development of a custom syntax for aggregate operations, different from standard SQL, that eases the extraction of a record window from the local buffer. Consider the following example: `SELECT AVG(temperature, 10 SAMPLES)`. This query excerpt instructs the LLQE to extract the last 10 temperature samples from the local buffer, compute the average, and output the result. A similar syntax allows the selection of a record window using time intervals (e.g., `SELECT AVG(temperature, 1 HOUR)`).

The **Execution Conditions Section** defines the rules to establish if a certain logical object should participate in the query. The EXECUTE IF clause introduces the conditions that must be satisfied by an FPC in order to be admitted for query execution; these conditions can refer both to the current value of an attribute and to the existence of attributes and events (e.g., `EXECUTE IF temperature > 15 AND EXISTS (temperature)` forces the execution of a query on all the nodes that have a temperature sensor and that are currently sensing more than 15 degrees). The middleware component responsible for the evaluation of the termination condition is the FPC Registry (see Section 3.1.2). This section of the query is optional and, if not specified, all the Logical Objects in the system will be

involved in query execution. For this reason, a precise semantics for null values management has been introduced in order to deal with situations in which the value of an attribute is required and the logical object does not expose it. The EXECUTE IF can be optionally complemented with a *REFRESH* clause that specifies when the execution condition has to be reevaluated to update the list of nodes involved in the evaluation of a query.

The *PILOT JOIN* is another clause designed to determine the set of Logical Objects on which a Low-Level Query has to be executed. Differently from the EXECUTE IF, the PILOT JOIN allows the definition of an execution condition based on records produced by a second independent query. This clause adds the possibility of designing complex execution conditions and enables the user to employ the output of one or more unrelated sensors to trigger the execution of a query (refer to Section 4.2 for an example of the PILOT JOIN). By means of the PILOT JOIN, it is also possible to implement a basic form of *context-aware data tailoring* [4], [3].

After the evaluation of the execution condition section, a logical object is possibly charged with executing a certain Low-Level Query. After the query is started, only the sampling and the data management sections remain active until the termination condition becomes true or an explicit termination of the query is required. Thus, a running query can be conceptually thought of as a couple of threads that, respectively, act as a producer (that inserts data into the local buffer according to the sampling section) and as a consumer (that produces output records manipulating the local buffer content, according to the data management section). As said before, both the previous threads can be activated periodically or when an event happens.

The **Termination Conditions Section** is the last block composing a low-level statement and is introduced by the *TERMINATE AFTER* clause. Although queries submitted to a pervasive system often have to be executed in a continuous fashion and are manually stopped, PerLa supports a mechanism to optionally set the execution lifetime in the query itself, either in terms of number of selections (`TERMINATE AFTER 3 SELECTIONS`) or runtime (`TERMINATE AFTER 5 HOURS`). This can be useful when a one shot query has to be executed (e.g., to monitor the current state of a logical object or to sample a single value from a specific sensor) or when the monitoring period is known a priori.

Consider a system in which the nodes are some ad hoc boards placed in a building and provided with temperature or pressure sensors. Suppose that a user is interested in retrieving, every 10 minutes, the maximum temperature reached in a given room by sampling each sensor once a minute. Table 3a shows an example LLQ that has exactly this behavior.

The *execution conditions section* of the query specifies that the statement must be executed only on the Logical Objects that expose a `temp` attribute, i.e., on the devices able to sense temperature values. Moreover, we restrict the selection to only those devices that are placed in room 3 (we assume the existence of a `room` static attribute, set at deployment time on every node of the sensing network, as mentioned in Section 3.1.1). The *sampling section* requires the execution of a time-based sampling, with a period of one minute. Also the *data management section* is activated with a time-based semantics, and the computation of the maximum temperature value sensed during the last 10 minutes is required. This is a *continuous query* since no termination condition is specified; thus, an explicit stop command is required to halt its execution. Note that the analysis of the query allows one to bound the ideally infinite buffer that acts as the interface between the sampling and the data management sections; for example, a history of 10 minutes is enough to produce the correct results in the considered query.

Table 3b shows an example of event-driven query. Its goal is to report, after 10 minutes, how many times a given RFID tag was sensed by each reader. Whenever the `lastReader-Changed` event of the considered tag is raised, a sampling operation is performed and a record reporting the RFID reader identifier is appended to the local buffer. The data management section is activated only once (due to the termination condition) with a time-based semantics, and it aggregates and counts the data contained in the local buffer.

4 Case Studies

In this section, two case studies in which the PerLa system has been employed are presented. The first test bed application is related to the monitoring of a mountain that is subject to rockfall phenomena. This example is presented focusing on the middleware rather than on the language: The hardware architecture and the definition of the FPCs are discussed. The second test bed application is related to the monitoring of the wine production process, and it is presented especially focusing on PerLa language features: the application context, the relevant physical measures, the logical object interface, and some examples of typical queries are reported.

4.1 Rockfall Monitoring

In this section, we briefly introduce a real geophysical monitoring application, which has been chosen for the first deployment of the middleware described in this paper. The project consists in the monitoring of Monte San Martino mountain, a mountain located near Lecco, Italy, in order to predict rockfall events (that have already happened on that mountain in the past). The main goal of the project is to set up a model, based both on historical data and real-time measurements, able to predict rockfall events early. Thus, a certain set of sensors has been deployed on the San Martino face in order to measure some physical parameters, like accelerations, noises, and inclinations. The main constraint of this application is related to power management: When deployed, the nodes cannot be easily reached anymore because they are placed on dangerous points. As a consequence, the network nodes must be "low power" and equipped with solar panels. The case study discussed in this section is a particular embedded system in which the physical devices were ad hoc designed and created by our staff.

The Monte San Martino monitoring system has been working since the end of April 2010, with a duty cycle of 24 hours/day. PerLa managed more than 2,500,000 measurement records from 30 sensors, of six different types, in six months with no interruption. A large saving of coding effort

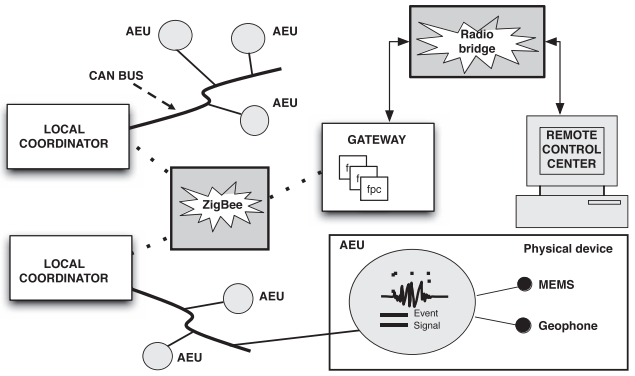

Fig. 3. HW architecture for rockfall monitoring.

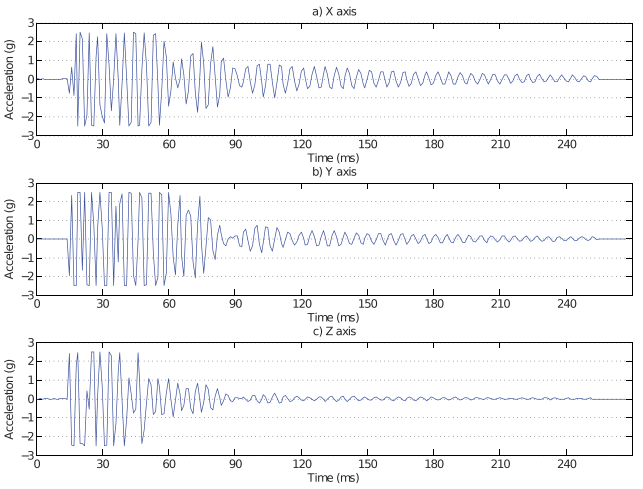

Fig. 4. Microacoustic burst signal acquired by DSPics.

was obtained thanks to the flexibility of PerLA in adding new types of devices which had not been foreseen at the outset.

In the following, the employed hardware and software architecture is described.

4.1.1 Hardware Architecture

Fig. 3 shows the whole architecture that has been designed for this application. The *Acquisition and Elaboration Units* (AEU) are *DSPic33f*-based boards [50], equipped with an accelerometer or with a geophone. Each of these units continuously monitors physical vibrations, and preprocesses the sampled signals in order to detect some events that could be related to the monitored rockfall phenomenon, i.e., the formation and organization of fractures inside rocks. Fig. 4 presents typical microacoustic signals acquired by the DSPic nodes. In particular, the three diagrams show the accelerations over the X, Y, and Z axes.

Many of the sensed events are false positives; in fact, an event is useful only when it is simultaneously detected by more than one node. For this reason, a certain number of sensing units are connected to a CAN-bus channel since a wireless network does not allow to reach the required level of synchronism. Data sampled by the sensing units are collected by a local coordinator, connected to the same CAN-bus. This coordinator is a "low power" Linux-based embedded board that is not powerful enough to host a Java virtual machine. The function of the local coordinator is to act as a master on the CAN-bus and to route collected data toward the gateway through a *ZigBee* [51] network. The gateway is a powerful Linux-based embedded board and it is the only device that will be installed in a safe area of the mountain face. It is connected to a remote control center through a TCP/IP radio bridge.

Finally, the system also manages data from solar cells power sources. These devices are "black boxes" as to their internal structure, and therefore PerLa creates a wrapper to manage the monitoring data.

The goal of the application is to manage collected data and to control involved devices entirely from the control center. The application level software that is provided to the users exploits PerLa features to collect required data from the system.

4.1.2 Software Architecture

The *AEUs* are the physical nodes for which an FPC is generated. This FPC exposes: the ID, the latitude, and longitude of the node (static attributes); the burst detection notification (event); the parameters of the last detected event such as initial time, duration, maximum amplitude, first Fast Fourier Transform *(FFT)* coefficients (nonprobing dynamic attributes); and the current acceleration values on the X, Y, and Z axes (probing dynamic attributes).

Note that the defined FPCs can be queried using both the event-based and the time-based sampling mode. In the first case, the list of detected events can be retrieved, while in the second case the inclination of the node can be periodically monitored (relying on current acceleration values). The FPC components and the FPC Factory are placed on the gateway; in fact, it is the nearest device able to run a JVM and connected to the control center via TCP/IP (through the radio channel).

AEU boards run the FreeRTos operating system. The program deployed on these devices performs a real-time digital filtering on the 2 KHz MEMS signal and executes the event detection algorithm. Note that this computation must be executed at this level because the transmission of the whole sampled signal as a data stream is too expensive in terms of bandwidth and power consumption. The AEU program implements a simple communication protocol on the CAN-bus channel in order to modify some parameters (e.g., thresholds of the event detection algorithm) and to return sampled data. It should be noted that the software deployed on the AEU can be thought of, from the middleware point of view, as the low level API provided by the device vendor.

When a user requires the current value of the inclination attribute, the FPC builds up a request message that is sent on the virtual channel (i.e., the ZigBee link). The sampled data returned by the sampling routine are then sent back to the FPC over the virtual channel. Finally, the FPC performs the conversion and the normalization of the sampled inclination value, based on the information contained in the self-configuration file. Similarly, when an actuation query is submitted on the FPC, the parameters and the values to be set are sent on the virtual channel by the middleware code. The only routine that must be written by the device vendor is the C code to send a proper message on the CAN-bus in order to update the parameters of the algorithm running on the AEU.

```
SET MemsXGain=127, MemsYGain=127, MemsZGain=127, ....,
  FilterTapsX = NEW(CONSTANTVECTORINTEGER,
    "0xFE02, ....")
EXECUTE IF ID = 1 OR ID = 2 ...
```

Fig. 5. Actuation query example.

All the devices (DSPic, Local Coordinator, and Gateway) can be fine-tuned by modifying the corresponding parameters. In particular for the local coordinator and gateway, the duty-cycle can be changed at runtime. Moreover, the acquisition, the filtering, and the detection algorithm of the DSPic are parameterized to allow an in-situ deep investigation of the microacoustic burst. PerLa offers the actuation queries to achieve this goal. A simplified version of an actuation query for DSPics is presented in Fig. 5; the `MemsXGain`, `MemsYGain`, `MemsZGain` are the parameters used to change amplifier gains of the three MEMS channels; `FilterTapsX` represents the coefficients of the digital filter used on the MEMS X signal channel (note that the filter is a 128-coefficient FIR filter).

4.2 Wine Production Process Monitoring

The test bed application discussed in this section refers to the automation of a large wine production farm, from the vineyard to the table. This is one of the case studies of ARTDECO, a large project funded by the *Italian University Ministry*. Some parameters of the production cycle must be kept under strict control, both to guarantee the quality and the integrity of the product and to ensure complete traceability of all the wine lots until the final labeling phase. The first monitoring is performed through humidity, temperature, and chemicals sensors, placed in the vineyard and connected through a wireless network. Moreover, the men who work in the vineyard are equipped with PDAs, running a worker information system: These PDAs can also be used to sense temperature and position.

The PerLa Language can handle all these nodes by interacting with their Logical Objects that basically expose the node unique identifier, the device type, the current powerLevel, `locationX` and `locationY` attributes, a temperature value, and, optionally, a humidity value. Note that the attributes revealing the current position are static for the wireless nodes (since they are placed in a known point during the system setup), while they are dynamic for the PDAs.

Suppose that the monitoring of the environment parameters (temperature and, if available, humidity) in the vineyard is required. The low-level statement shown in Table 4a can be used to continuously sample, once per minute, the temperature and the humidity in the whole vineyard and to insert the results into a `Monitoring` stream. The EXECUTE IF clause requires the execution of the query on all the nodes currently placed in the yard and having a temperature sensor on board, but it does not restrict the query execution to wireless nodes only. Therefore, suppose that a worker is in the yard with his PDA and that device will execute the query exactly as the wireless nodes do (with the only difference that `NULL` values will be produced for the humidity since the corresponding sensor is unavailable). The REFRESH clause forces the EXECUTE IF condition to be evaluated again every 10 minutes in order to update the list of nodes running the query. Suppose now that the user is not interested in continuously retrieving the sampled data, but only in receiving some alarms when the monitored parameters exceed established thresholds. As an example, consider the frost phenomena that must be certainly avoided in a vineyard. It is known that a temperature under 5 degrees combined with a humidity over 75 percent is an environmental condition that can cause a vineyard to freeze; the query shown in Table 4b can be used to signal a suitable alarm. Note that the frost condition is evaluated on the average of the last 10 minutes samples in order to remove incidental noise.

TABLE 4
Wine Production Process Monitoring—Queries

a)
```
CREATE OUTPUT STREAM Monitoring
 (nodeId ID, temperature FLOAT, humidity FLOAT,
 locationX FLOAT, locationY FLOAT)
AS LOW:
  EVERY ONE
  SELECT ID, temperature, humidity, locationX, locationY
  SAMPLING
   EVERY 1 m
  EXECUTE IF EXISTS (temperature) AND
    is_in_Vineyard(locationX, locationY)
  REFRESH EVERY 10 m
```

b)
```
CREATE OUTPUT STREAM FrostAlarm (nodeId ID, ts TIMESTAMP)
AS LOW:
  EVERY ONE
  SELECT ID, TIMESTAMP
  HAVING (AVG (temperature, 10 m) < 5) AND
    (AVG (humidity, 10 m) > 0.75)
  SAMPLING
    EVERY 1 m
  EXECUTE IF EXISTS (temp) AND EXIST (humidity) AND
   is_in_Vineyard(locationX, locationY)
```

c)
```
CREATE OUTPUT STREAM LowPoweredDevices (sensorID ID)
AS LOW:
  EVERY ONE
  SELECT ID
  SAMPLING
    EVERY 24 h
  WHERE powerLevel < 0.15
  EXECUTE IF deviceType = "WirelessNode"
```

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

e)
```
CREATE OUTPUT STREAM OutOfTemperatureRangePallets
   (palletID ID)
 AS LOW:
  EVERY 10 m
    SELECT ID
      SAMPLING EVERY 10 m
    WHERE temperature > 18
    PILOT JOIN TrucksPositions ON baseStationID =
      TrucksPositions.linkedBaseStationID
```

f)
```
CREATE OUTPUT STREAM BottlesEnteredInCellar
  (bottleId ID, ts TIMESTAMP)
AS LOW:
  EVERY ONE
  SELECT ID, TIMESTAMP
  SAMPLING
    ON EVENT lastReaderChanged
  WHERE lastReaderId = "CellarX_ReaderId"
```

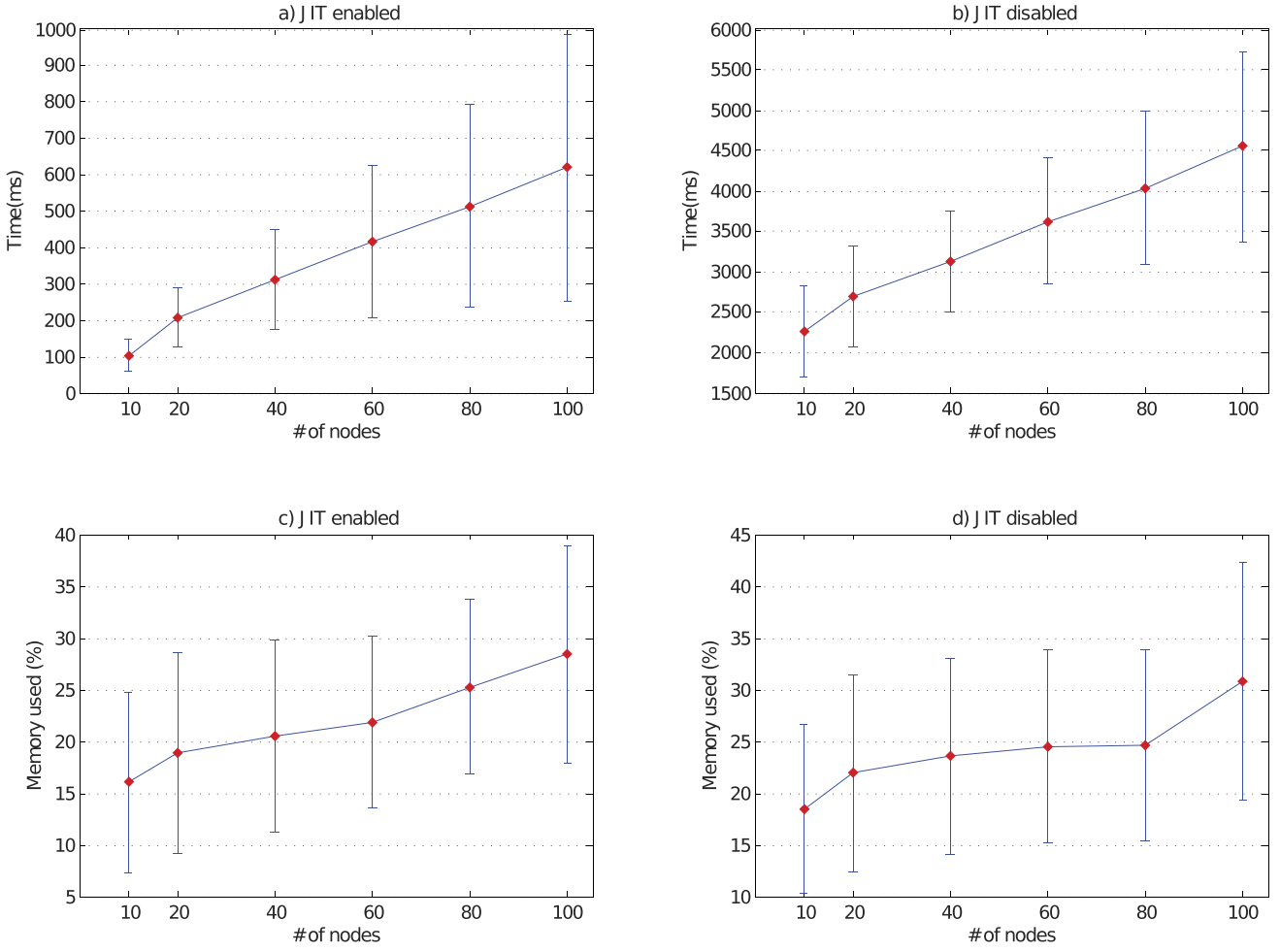

Fig. 6. Data processing time and memory usage in the San Martino rockfall monitoring application.

An important feature of the PerLa Language is its ability to query the network state as "ordinary" 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 and to allow the substitution of the batteries before the nodes become inactive. Table 4c shows a statement that can be used exactly for this purpose.

The transport is another important and critical phase in the wine life cycle that should be controlled in order to avoid sudden changes of temperature of the transported bottles. To support the monitoring of this phase, every truck is equipped with a GPS and with a base station. Moreover, each pallet has a temperature sensor used to sense the temperature of the contained bottles. The query reported in Table 4e produces the list of pallets whose temperature exceeded a certain threshold while the truck was traveling through a zone considered particularly critical. In that 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 (see Table 4d). Note that the interface of the Logical Objects wrapping GPS devices must have an attribute (`linkedBaseStationId`) that retrieves the ID of the base station reinstated on the same truck (this is a static attribute, whose value is defined at network deployment time).

The last example that is presented is relative to the management of the bottles stored in the cellars. They are all equipped with a low-frequency RFID tag that allows one to completely trace the history of the bottle. An RFID reader is mounted on the door of each cellar; thus, it is possible to know exactly when a bottle entered and left the cellar. Moreover, temperature and humidity sensors are employed to constantly monitor the cellar temperature in a similar way as explained for the vineyard. Thus, the temperature and the humidity fluctuations a bottle was subject to can be exactly traced. Table 4f shows an LLQ that can be used to monitor the bottles entering in a given cellar.

5 System Evaluation

This section provides a twofold evaluation of the PerLa system: first, an assessment of the performance of the middleware; then, an analysis of the benefits in terms of development speed achieved through the use of PerLa.

5.1 Performance

Fig. 6 summarizes the resources required by the PerLa system to convert and process messages coming from the devices described in Section 4.1.

This information was gathered using a custom-made benchmark which mimics a homogeneous network of DSPic Acquisition and Elaboration Units. The simulation was performed on a PC equipped with a 2.40 GHz Intel Core 2 Quad CPU Q6600 and 4 GB of RAM, running a 64 bit

Linux Kernel version 2.6.32. The JVM used is the 64 bit Sun Java virtual machine version 1.6.0 20. The number of DSPic nodes in the simulated network ranged from 1 to 100, in increments of 10 units.

While running the benchmark, each node of the simulated network sends a continuous stream of 1,000 messages. Every message has a fixed size of 7,450 bytes, and contains a 1 sec waveform that represents the three axes signal sample recorded during a microseismic activity, along with some additional information (e.g., timestamp, peak value, noise ratio, etc.).

Both execution time and memory occupation were recorded to evaluate the scalability of the system. Diagrams a, b, c, and d of Fig. 6 present the results observed from the same simulation, executed with and without *Just In Time* (JIT) Java compilation optimization.

The experiments a and b of Fig. 6 shows that the average message processing time scales linearly as we increase the number of nodes. This behavior is determined by the intrinsic multithreaded architecture of PerLa. In fact, all FPCs of the system run concurrently (each of them is executed in a separate thread). Moreover, all Java classes shared among different FPCs were designed to foster parallel execution. In Fig. 6a, the runtime JIT optimization greatly reduces the average execution time, but a "cache"-like effect concurs in increasing the observed standard deviation. This phenomenon is absent in Fig. 6b, which displays the results of a simulation performed without JIT optimization.

Figs. 6c and 6d summarize the memory occupation statistics collected during the execution of our simulation. Both charts show that the total memory occupation increases as the benchmark adds new nodes, and reaches about 30 percent of the total memory available to the JVM (850 MB). The amount of variance experienced during these experiments is due to the effects of the Java Garbage Collector, which periodically redeems the memory held by unused objects. The reader should also note that the maximum memory occupation is reached more slowly when JIT is disabled, due to the reduced performances of the JVM under these conditions. In fact, the execution speed does not affect the FPCs alone, but also the threads that generate DSPic messages as well (memory occupation is greatly influenced by the number of data messages queued in the system during the benchmark).

It is worth noting that this simulation recreates a "stress-case" scenario, designed to evaluate the performance of the system under heavy workload. In the deployment described in Section 4.1, the total number of sensors is 30, and each of them sends one message every 5 minutes. The message throughput simulated in the benchmark application, due to bandwidth limits imposed by the different communication channels utilized, would not even be achievable in the real-life case study. Moreover, the particular 7,450 bytes message sent by DSPic nodes in this rockfall monitoring sensor network is unusual for WSNs applications. For comparison, in ZigBee applications, data messages are usually limited to 80 bytes.

5.2 Benefits

As said in Section 3, PerLa is a tool for managing sensing networks, designed to support and expedite both low-level and high-level development processes.

The declarative, SQL-like PerLa language enables application developers to retrieve data from the sensing network with ease and simplicity. No knowledge regarding the specific hardware and software modules equipped on the sensing nodes is needed to write a PerLa query. This feature allows for a stronger separation among high-level applications and the acquisition network. Moreover, the PerLa language supports and promotes low-level heterogeneity since differences existing among the devices employed in the underlying sensing network are concealed by the middleware.

As mentioned earlier, PerLa provides different tools to support developers and users working on the lower levels of a sensing network. The integration of new typologies of devices in the PerLa middleware does not necessitate the explicit development of software drivers or java classes. Node developers are only required to describe the capabilities and features of their new hardware in a XML device description file; all the code needed to communicate and interface with the sensing nodes is generated at runtime by the middleware itself (see Section 3.1.2 for further details).

A brief evaluation of the effectiveness of this feature can be made by comparing the lines of code generated by PerLa versus the length of a device descriptor used in the Monte San Martino monitoring system. About 2,000 lines of procedural code were automatically assembled by the middleware to communicate with the Acquisition and Elaboration Units described in Section 4.1.1. This figure is a good approximation of the code needed to integrate the same device on other systems (e.g., to build a GSN wrapper), and comprises all software components needed to marshal and unmarshal data messages, manage the communication with the physical sensing devices, and assemble the output records needed by the query executors. The length of the corresponding XML device descriptor, in comparison, was only 57 lines long.

PerLa proved to be of invaluable help during the development of both case studies of Section 4. Substantial reductions in development time were achieved through the extensive use of the aforementioned middleware features instead of more customary approaches which would have required the development of an ad hoc managing application. An assessment of these time savings can be obtained with the analysis of a particular situation which occurred during the development of the Monte San Martino monitoring system: Due to a refinement in the sampling procedure, many of the data structures employed by the acquisition nodes to communicate with the PerLa system changed. PerLa allowed device designers to quickly mirror these changes to the middleware by simply modifying the XML device descriptor, a task that required only a fraction of an hour. No query had to be rewritten due to this alteration, and no code had to be adapted by hand to align the middleware with the new message format. As noticed in Section 4.1, large savings of coding effort were also obtained thanks to the flexibility in adding new device types which had not been envisaged at the beginning of the project.

Of the 20 man months required for the complete development of the monitoring system installed on the San Martino mountain, only a couple of weeks were spent in activities required to customize the PerLa system for the specific application (i.e., writing queries, XML descriptors,

and, occasionally, components needed to communicate with new physical media and protocols).

6 Conclusions and Future Work

In this paper, we presented the design and the implementation of PerLa, a language and infrastructure for data management in pervasive systems, mainly oriented to monitoring applications, but also suitable as a support substratum for the deployment of autonomic systems. The paper is mainly focused on the issues related to the heterogeneity of the different devices composing a pervasive system: This aspect is investigated both at data management and at physical integration levels. The presented solution aims at handling these issues using a nontrivial approach.

The PerLa architecture is composed of two main elements. A declarative SQL-like language has been designed to provide the final user with an overall homogeneous view on the whole pervasive system. The provided interface is simple and flexible, and it allows users to completely control each physical node, masking the heterogeneity at the data level.

A middleware has been implemented in order to support the execution of PerLa queries, and it is mainly charged to manage the heterogeneity at the physical integration level. The key component is the Functionality Proxy Component, which is a Java object able to represent a physical device in the middleware and to take the device's place whenever unsupported operations are required. In this paper, we have not investigated some optimizations that can improve the performance of the middleware and the expressiveness of the language. Our future work will be focused on integrating intelligent in-network processing protocols [52] in the middleware and on extending the language semantics with compression and data mining operators.

We are also going to extend both the language and the middleware to support context-based applications. The basic idea is to change the behavior of the surrounding network (e.g., by injecting an actuation query in the nodes) depending on the actual context. From the language point of view, although the pilot join operation is a first attempt to support context-based queries, we shall extend PerLa with specific context management operators. Building upon existing data retrieval capabilities of PerLa, these new language features will be able to support context management and context-aware applications (see [3] and [4]).

From the middleware point of view, a context can be seen like an agent which, depending on the current validity condition, automatically injects different sets of queries into the system. The condition is periodically checked by a new component, the Context Manager, which also spawns or disables context agents.

Finally, we plan on integrating some energy-aware data reduction facilities into the language in order to reduce the network traffic [53].

Acknowledgments

This research has been partially funded by the following research projects: Politecnico di Milano PROMETEO P2ICT Lab, Italian MIUR-FIRB Art-Deco, European Commission Programme IDEAS-ERC Project 227077-SMScom.

References

[1] ARTDECO, project web page, http://artdeco.elet.polimi.it, 2011.
[2] C. Batini, S. Ceri, and S. Navathe, *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin Cummings, 1992.
[3] C. Bolchini, C.A. Curino, G. Orsi, E. Quintarelli, R. Rossato, F.A. Schreiber, and L. Tanca, "And What Can Context Do for Data?" *Comm. ACM,* vol. 52, no. 11, pp. 136-140, Nov. 2009.
[4] C. Bolchini, F.A. Schreiber, and L. Tanca, "Data Management" *Mobile Information Systems,* B. Pernici, ed., pp. 155-175, Springer Verlag, 2006.
[5] F.A. Schreiber, R. Camplani, M. Fortunato, M. Marelli, and F. Pacifici, "PerLa: A Data Language for Pervasive Systems," *Proc. Sixth Ann. IEEE Int'l Conf. Pervasive Computing and Comm.,* pp. 282-287, 2008.
[6] M. Weiser, "The Computer for the 21st Century," *Scientific Am.,* vol. 265, no. 3, pp. 66-75, Sept. 1991.
[7] T. Kindberg and A. Fox, "System Software for Ubiquitous Computing," *IEEE Pervasive Computing,* vol. 1, no. 1, pp. 70-81, http://dx.doi.org/10.1109/MPRV.2002.993146, Jan. 2002.
[8] A. Messer, H. Song, D. Cheng, and S. Gibbs, "A Classification of Pervasive System Software," *Common Models and Patterns for Pervasive Computing Workshop,* 2007.
[9] H. Salem and M. Nader, "Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks," *IEEE Distributed Systems Online,* vol. 7, no. 3, p. 1, Mar. 2006.
[10] M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, 1999.
[11] D.J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Data Eng. Bull.,* vol. 32, no. 1, pp. 3-12, Mar. 2009.
[12] C.A. Mattmann, S. Malek, N. Beckman, M. Mikic-Rakic, N. Medvidovic, and D.J. Crichton, "Glide: A Grid-Based Light-Weight Infrastructure for Data-Intensive Environments," *Proc. Advances in Grid Computing—European Grid Conf.,* pp. 68-77, 2005.
[13] N. Davies, A. Friday, and O. Storz, "Exploring the Grid's Potential for Ubiquitous Computing," *IEEE Pervasive Computing,* vol. 3, no. 2, pp. 74-75, Apr.-June 2004.
[14] L.W. McKnight, J. Howison, and S. Bradner, "Wireless Grids: Distributed Resource Sharing by Mobile, Nomadic, and Fixed Devices," *IEEE Internet Computing,* vol. 8, no. 4, pp. 24-31, July/Aug. 2004.
[15] C. Hartung, R. Han, C. Seielstad, and S. Holbrook, "Firewxnet: A Multi-Tiered Portable Wireless System for Monitoring Weather Conditions in Wildland Fire Environments," *Proc. Fourth Int'l Conf. Mobile Systems, Applications and Services,* pp. 28-41, 2006.
[16] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," *Proc. ACM Int'l Workshop Wireless Sensor Networks and Applications,* Sept. 2002.
[17] G.W. Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh, "Monitoring Volcanic Eruptions with a Wireless Sensor Network," *Proc. Second European Workshop Wireless Sensor Networks,* Jan. 2005.
[18] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.S. Peh, and D. Rubenstein, "Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with Zebranet," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* vol. 37, no. 10, pp. 96-107, Oct. 2002.
[19] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Systems,* vol. 30, no. 1, pp. 122-173, 2005.
[20] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "Tag: A Tiny Aggregation Service for Ad-Hoc Sensor Networks," *ACM SIGOPS Operating Systems Rev.,* vol. 36, no. SI, pp. 131-146, http://dx.doi.org/10.1145/844128.844142, 2002.
[21] P. Levis et al., "TinyOS: An Operating System for Sensor Networks," *Ambient Intelligence,* pp. 115-148, Springer, 2005.
[22] D. Chu, L. Popa, A. Tavakoli, J.M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The Design and Implementation of a Declarative Sensor Network System," *Proc. Fifth Int'l Conf. Embedded Networked Sensor Systems,* pp. 175-188, 2007.
[23] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein, "Entirely Declarative Sensor Network Systems," *Proc. 32nd Int'l Conf. Very Large Data Bases,* pp. 1203-1206, 2006.
[24] "Sword—Business Service," Siemens, http://webdoc.siemens.it/CP/SIS/Press/SWORD.htm, internal comm., 2010.

[25] K. Aberer, M. Hauswirth, and A. Salehi, "The Global Sensor Networks Middleware for Efficient and Flexible Deployment and Interconnection of Sensor Networks," technical report, submitted to ACM/IFIP/USENIX Seventh Int'l Middleware Conf., 2006.

[26] K. Aberer, M. Hauswirth, and A. Salehi, "A Middleware for Fast and Flexible Sensor Network Deployment," *Proc. 32nd Int'l Conf. Very Large Data Bases,* pp. 1199-1202, 2006.

[27] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST: A Protocol for Integrating Sensor Networks into the Internet," *Proc. REALWSN,* 2005.

[28] R. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," PhD dissertation, Univ. of California, 2000.

[29] "JMS—Java Message Service,"Oracle, http://java.sun.com/products/jms/, 2011.

[30] "RMI—Remote Method Invocation," Oracle, http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, 2011.

[31] "CORBA, the Common Object Request Broker Architecture," OMG, http://corba-directory.omg.org/, 2011.

[32] J.P. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," *Proc. Third Working IEEE/IFIP Conf. Software Architecture,* pp. 29-43, 2002.

[33] M. Román, C.K. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces," *IEEE Pervasive Computing,* vol. 1, no. 4, pp. 74-83, Oct.-Dec. 2002.

[34] T. Kindberg and J.J. Barton, "Towards a Real-World Wide Web," *Proc. ACM SIGOPS European Workshop,* pp. 195-200, 2000.

[35] http://www.cascadas-project.org, CASCADAS project web page, 2010.

[36] S. Bindelli, E.D. Nitto, R. Mirandola, and R. Tedesco, "Building Autonomic Components: The Selflets Approach," *Proc. IEEE Int'l Conf. Automated Software Eng. Workshops,* pp. 17-24, 2008.

[37] D. Devescovi, E.D. Nitto, D. Dubois, and R. Mirandola, "Self-Organization Algorithms for Autonomic Systems in the Selflet Approach," *Proc. First Int'l Conf. Autonomic Computing and Comm. Systems,* pp. 1-10, 2007.

[38] G. Cugola and G. Picco, "REDS: A Reconfigurable Dispatching System," *Proc. Sixth Int'l Workshop Software Eng. and Middleware,* p. 16, 2006.

[39] "Drools—The Business Logic Integration Platform," JBoss Community, http://jboss.org/drools, 2010.

[40] A. Murphy, G. Picco, and G. Roman, "Lime: A Middleware for Physical and Logical Mobility," *Proc. Int'l Conf. Distributed Computing Systems,* pp. 524-533, 2001.

[41] "OSGi Service Platform Core Specification," O. Alliance, http://www.osgi.org/Download/Release4V41, 2011.

[42] D. Chakraborty, F. Perich, A. Joshi, and Y. Yesha, "Middleware for Mobile Information Access," *Proc. Fifth Int'l Workshop Mobility in Databases and Distributed Systems,* Sept. 2002.

[43] F. Perich, S. Avancha, D. Chakraborty, A. Joshi, and Y. Yesha, "Profile Driven Data Management for Pervasive Environments," *Proc. 13th Int'l Conf. Database and Expert Systems Applications,* Sept. 2002.

[44] S. Avancha, D. Chakraborty, H. Chen, L. Kagal, F. Perich, and A. Joshi, "Issues in Data Management for Pervasive Environments," *Proc. NSF Workshop Context-Aware Mobile Database Management,* Jan. 2002.

[45] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems," *IEEE Trans. Software Eng.,* vol. 31, no. 3, pp. 256-272, Mar. 2005.

[46] L. Golab and M. Özsu, "Issues in Data Stream Management," *ACM SIGMOD Record,* vol. 32, no. 2, pp. 5-14, 2003.

[47] "JAXB, Java Architecture for XML Binding," Oracle, http://java.sun.com/developer/technicalArticles/WebServices/jaxb/, 2011.

[48] "CAN Bus Specifications," ISO Standard 11898 1 2003, http://www.iso.org, 2011.

[49] F.A. Schreiber, R. Camplani, M. Fortunato, M. Marelli, and G. Rota, "Design of PerLa, a Declarative Language and a Middleware Architecture for Pervasive Systems," Technical Report 2010.9, ARTDECO R.A.11b, DEI, pp. 1-141, 2010.

[50] "DSPic, High-Performance 16-Bit Digital Signal Controllers," MicroChip, http://ww1.microchip.com/downloads/en/devicedoc/70155c.pdf, 2010.

[51] Z. Alliance, "Zigbee Specification," *ZigBee Document 053474r06, Version,* vol. 1, 2005.

[52] G. Cugola and M. Migliavacca, "A Context and Content-Based Routing Protocol for Mobile Sensor Networks," *Proc. European Conf. Wireless Sensor Networks,* pp. 69-85, 2009.

[53] C. Cappiello and F.A. Schreiber, "Quality- and Energy-Aware Data Compression by Aggregation in WSN Data Streams," *Proc. IEEE Int'l Conf. Pervasive Computing and Comm.,* pp. 634-639, 2009.

Fabio A. Schreiber is a full professor of databases and of technologies for information systems in the Department of Electronic and Information Engineering at the Politecnico di Milano. His current research interests include: data management in pervasive systems, database systems for context-aware applications, very small and mobile database design methodologies and applications. On these and other topics, he has authored 100 papers published in international journals and conferences. He is a life senior member of the IEEE.

Romolo Camplani received the DrIng degree in electronic engineering and the PhD degree in electronic and computer science from the Universita' degli Studi di Cagliari, Cagliari, Italy, in 2003 and 2008, respectively, and the MEng degree in embedded system design from the Universita' della Svizzera Italiana, Lugano, Switzerland. He is currently a postdoctoral researcher with the Politecnico di Milano, Milan, Italy. His research interests include wireless sensors networks, adaptive routing algorithms, and power-aware embedded applications.

Marco Fortunato received the DrEng degree in computer science engineering from the Politecnico di Milano, Milano, Italy, in July 2008. He is a software developer and has been collaborating with the Dipartimento di Elettronica e Informazione on the development of the PerLa project since 2007. His research interests include databases and data management in pervasive systems.

Marco Marelli received the DrEng degree in computer science engineering from the Politecnico di Milano, Milano, Italy, in December 2007. He is a software developer and has been collaborating with the Dipartimento di Elettronica e Informazione on the development of the PerLa project since 2007. His research interests include databases and data management in pervasive systems.

Guido Rota is a graduate student at the Politecnico di Milano, where he received the bachelor's degree with a thesis on "A Distributed Support System for Transportation Collision Management." He also works as an informatic consultant and software developer.