POLITECNICO DI MILANO
Facoltà di Ingegneria
Corso di Laurea Triennale in Ingegneria Informatica

Design and implementation of the Low Level Query environment for PerLa language

Relatore: Ch.mo Prof. Fabio A. SCHREIBER
Correlatore: Ing. Romolo CAMPLANI

Tesi di laurea di:
Diego VIGANO'
mat. 668589

Anno Accademico 2009/2010

Contents

List of Figures

Thesis objectives

This thesis aimed at the definition, design and implementation of the Low Level Execution environment of PerLa [5, 6] language, which is mainly charged of collecting data from the underlying pervasive system, preparing them for further computations at upper levels. As it will be explained in Subsection 3.2.2, this thesis found its location within logical level, and particularly in logical objects. In details, this thesis intended to accomplish the following goals:

- Achieve an overall design of the Low Level Query Execution, focusing on the complex aspects regarding the collection of data from the underlying pervasive system, since a complete design still, in fact, needed to be fulfilled. This objective included the implementation of those elements which are responsible to make data exchange possible between FPC and LLQ environment.

- Consolidation and design of the interface between the FPC and the Low Level Query Execution. The FPC was, in fact, designed in [7] and partially implemented in [8] but its implementation focused on the interactions with physical devices, while a well defined interface with the LLQ environment was still to be completely designed and achieved.

1 Introduction

1.1 Pervasive systems and Wireless Sensors Networks

Computing has, particularly in the last years, leapt off from the desktop to insinuate itself into everyday life, slowly making real the concept of *pervasive computing.* According to [9], this is only the final step of a series of changes in computing vision, started in the mid 1970s, when the PC first brought computer to the people, making the *personal computing* concept popular for the first time. Since then, the computing vision has encountered two important revolutionary changes that can be summarized as follows:

- *Distributed computing* (mid 1990s). Thanks to the advent of networking, it introduced the user to seamless access to remote information resources and communication
- *Mobile computing* (late 2000s). Integrating mobile technology into the Web, it has made possible to access information "anytime anywhere".

As mentioned before the last step of this change-in-vision is *pervasive computing*, whose goal is to fulfill the vision of an "all the time everywhere" access to information. Nowadays literature about this argument is vast, but, among many differences, one vision is shared between the experts of this sector: at their core, all models of pervasive computing must be composed of the following key components [9]:

- a set of **devices;**
- a **pervasive network**, used to link devices together;

- a **pervasive middleware**, playing a central role, making dialogue possible between the pervasive network and the final users;

- a **pervasive application** to allow the final user to interact with the system;

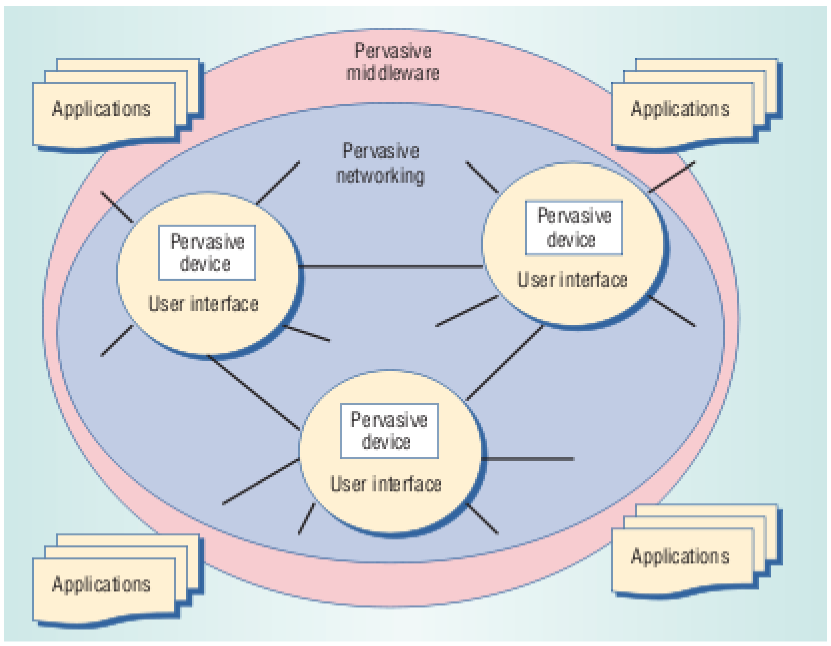


Figure 1.1: Pervasive system key components taken from [1]

Let's now focus on the model just presented. Maybe the most important application of this concept are the *Wireless Sensors Networks*. Such pervasive systems are composed of devices such as RFID tags, sensors, PDA and actuators linked together using a pervasive network (both cabled and wireless). *WSNs* have come under footlights, especially in the recent years, thanks to their rapid diffusion in many environments [1]:

- military applications (e.g.: battlefield surveillance, monitoring friendly and hostile forces, etc);

- environmental applications (e.g.: forest fire detection, flood detection, rockfall prevention [3]);

- home applications (home automation);
- health applications (remote patient surveillance);
- other applications (detecting car thefts, vehicle tracking system, etc).

1.2 Challenges of WSNs

According to many authors [1, 9] the WSNs present a series of challenges that every ad-hoc pervasive middleware should overcome and where the research is today focusing on:

Power saving support : this challenge is mainly related to the hardware adopted to realize the devices. In fact, from the energy point of view, the most delicate phase of a device is represented by the transmission of sampled data, that is the most power consuming activity among the all ones performable by a device. With power saving support we refer to the capability of the middleware to provide a series of algorithms and data structures that globally take care of the quantity of energy consumed by the device, and try to optimize the behaviour of the device to obtain the best performances. An example of such algorithms can be found in [10].

While, as said, power saving support is mainly related to hardware, there are several others challenges that concern the problem of how exploit the pervasive system in an efficent way, that are thus mainly related to software.

Heterogeneity support: different sensors networked together imply different technologies to be managed simultaneously. This requires the developer to make different protocols able to communicate each others. For example, today sensors (Motes, Jennic, Micaz to cite some examples) differ each others particularly for computational capabilities, power supplying methods (power cord, battery, solar power) and data gathering methods.

Configurability support: the network, especially if composed by hundreds of sensors, should be able to be easily configured, possibly at runtime if needed.

High level integration: the pervasive middleware that is going to manage the underlying (pervasive) network must expose an high level interface, allowing the user to easily write application or to integrate the WSN with information systems.

Flexibility and application reusability: Even if the developer could be able to succeed in managing heterogeneity, his solution could still be bound to the particular network he is dealing with. The risk lies in writing an application completely customized for a particular network, making it unusable for another network although slightly different.

Low Level support and Data gathering mechanisms: : These problems are somehow linked the heterogeneity support. Even if a middleware could be able to support heterogeneity, what about the programming effort needed to add a new device and make data exchange possible? With Low Level Support we indicate the capability of a pervasive middleware to "assist" the programmer in this effort, reducing the overall quantity of source code to be written, providing data structures, algorithms and, more in general, facilities that relieve the programmer of this task. Such type of "assistance" includes the data gathering mechanisms. With these terms we refer to the capability of the middleware of providing facilities to retrieve data from the pervasive system; in case of absence of these mechanisms the programmer is otherwise obliged to design and implement the code necessary to fulfill this fundamental task.

1.3 PerLa project: introduction and goals of the project

During the recent years, academic interests have taken into consideration the aforementioned challenges with many proposals explored in details in Chapter 2. PerLa (*PERvasive*

LAnguage) project finds its location in this set of proposals. Born within ART DECO project [11], the project aims at completely fulfilling the previous challenges. PerLa makes real the possibility of managing systems composed of sensors belonging to different technologies, allowing devices to be added and removed "on the fly", without affecting the whole system. Finally PerLa completely hides the complexity which derives from this goal to the final user.

The following chapters deeply analyze the differences between PerLa and other projects in order to underline the strengths points of PerLa compared to the other projects. In Chapter 3 the internal structure of PerLa will be presented, in order to show where this thesis finds its location within PerLa project.

2 State of the art

As said before, PerLa is not the only project aiming at undertaking the challenges exposed in Chapter 1. PerLa design started, during its first phases, searching for the potential limitations of other projects. Among the broad collection of such projects, the concentration was focused on the followings:

TinyDB [12]: this is the first and the most famous proposal to manage pervasive systems. Its most important feature lies in the view of the pervasive system as a database, allowing the user to query the system using a normal query language. Since this approach has been demonstrated being efficient, PerLa completely shares such vision. But TinyDB is still limited in dealing with homogeneous devices, and thus only partially solving the problems exposed in Section 1.2.

GSN [13]: GSN is a scalable, lightweight system which can be easily adapted, even at run time, to new type of sensors, thus allowing the dynamic reconfiguration of the system. Heterogeneity is supported, but it completely lacks of Low Level software support.

DSN [14]: DSN uses a completely different approach compared to the other middlewares proposed. The whole system, in fact, has been built using the declarative language Snlog (a dialect of Datalog), used both for the data acquisition and for the network and transmission management. This project lacks the important support of heterogeneity, fundamental in today WSNs.

SWORD [15]: Developed by SIEMENS, SWORD is a solution to control remotely a set of heterogeneous devices. The sensors (placed on the technological devices to be monitored) communicate with a central station where the SWORD application is running, allowing the final user to continuously monitor the situation remotely and to intervene if necessary. Also this project lacks of Low Level Software and, moreover, it doesn't provide data gathering mechanism since this task is left to the user.

As it can be seen from Table 2.1 many of these projects present some lacks that PerLa completely supports.

	TinyDB	***GSN***	***DSN***	***SWORD***	***PerLa***
Heterogeneity support	✗	✓	✗	✓	✓
Data gathering mechanisms	✓	✗	✓	✗	✓
High level integration	✓	✓	✓	✓	✓
Flexibility & Reusability	✗	✓	✗	✓	✓
Configurability support	✗	✗	✗	✗	✓
Low Level SW support	✓	✗	✓	✗	✓
Power saving support	✗	✗	✗	✗	✓

Table 2.1: Features supported by languages for WSNs.

As mentioned in Chapter 1 the challenge to achieve heterogeneity support is strictly linked to the concept of Low Level Software Support. It's now worth to analyze this relationship, with the help of the following figure:

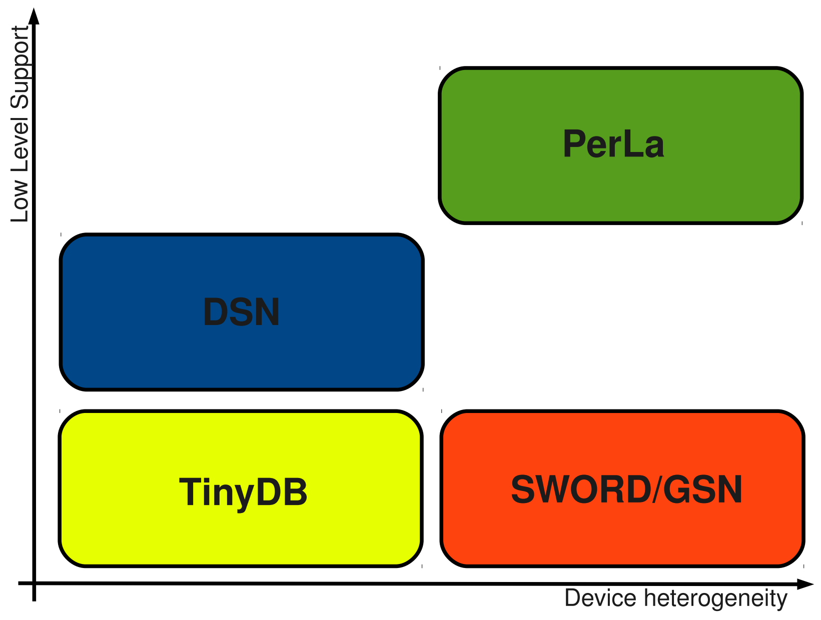


Figure 2.1: PerLa vs. others projects. A comparison

What emerges from this figure is that being able to support heterogeneity doesn't always imply to be able to do the same with Low Level Software support. The reader might remember the physical principle affirming that, when dealing with two different physical properties, the more precise one property is measured the less precisely the other will be[1]. Although heterogeneity and Low Level software support are not physical properties, this principle can be used as a metaphor to firmly understand that struggling to completely achieve one of the two can lead to fully unsupport the other and viceversa. This was the case of DNS, GSN, SWORD and TinyDB projects that decided to concentrate the efforts only on one of the two features, as shown in Figure 2.1. PerLa project is, instead, able to support both of them: such important feature will be fully described in the following chapters.

[1]The *Heisenberg uncertainty principle*

The dissertation continues now focusing on PerLa architecture, explaining how all the previously mentioned challenges are fulfilled.

3 PerLa architecture

This chapter focuses on the PerLa architecture that allows to fulfill all the challenges exposed in Chapter 1. The dissertation follows a top-down approach starting from looking at PerLa from the most general point of view. In this way, we can affirm that PerLa is composed by two interfaces, as exposed in the following figure:

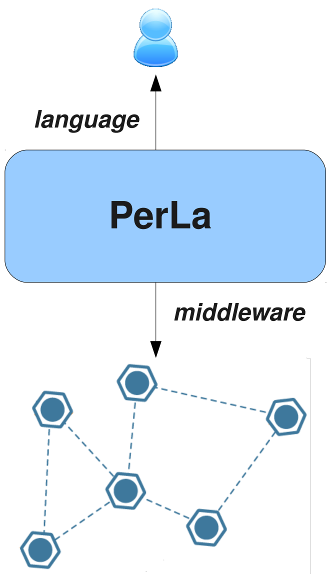


Figure 3.1: PerLa interfaces

- a "high level" interface for the final user, through the definition of a *language* able to manage the pervasive system;

- a "low level" interface to dialogue with the multitude of devices (and technologies) today available, achieving also the objective of relieving the programmer from the arduous task of managing the different technologies of such devices. This goal is fulfilled with the implementation of a *middleware.*

Following the top-down approach it's now worth to go deeper into details, focusing on these two, briefly introduced, interfaces and how their combined action allows PerLa to fulfill all the challenges exposed in Section 1.2.

3.1 The high level interface (language)

The *language* allows to manage the pervasive system and to declare how the user intends to exploit the WSN. It particularly allows the user to specify:

- **What** information must be retrieved from the WSN
- **When** and **how** such information must be retrieved
- **Where** the desired information should come from (i.e: to choose which sensors should be used, if the whole set composing the WSN or only a part of them satisfying a particular characteristic)

Since PerLa language was firstly designed [2], the objective of semantic power just exposed has required the design of a syntax that should have to be easy, fast to write and understandable to the final user. At the same time the language needed to be powerful: PerLa language was in fact designed trying to support both "standard" queries (i.e. queries written to retrieve some information from the WSN), and actuation queries (i.e. queries capable to set some parameters on sensors). Using an informatics paradigm it can be said that PerLa is expected to be able to "read" and "write" on the pervasive system. Moreover, since PerLa shares the vision of pervasive system as a database, the syntax is SQL-like although enriched with a series of operators and clauses. In order to achieve the required semantic power, the language was split in three logic parts:

- *Actuation Language,* allowing the user to set a number of parameters on one or more devices;
- *Low Level Language,* defining the behaviour of a single device (or a group of them), specially focusing on sampling operation and some aggregation operators;
- *High Level Language,* allowing the manipulation of data stream originated from low level queries.

It's immediate to understand that each language defines a certain number of operators and generates a proper query type:

- Actuation Queries (AQ)
- Low Level Queries (LLQ)
- High Level Queries (HLQ)

PerLa queries are then composed of a subset of the aforementioned three queries written in their proper languages, leaving to the user the power and flexibility to choose how the pervasive system can be exploited. In the following subsections a description of each query type (and its correspondent language) is given. Finally, in Subsection 3.1.3, a complete example of a quite complex PerLa query is shown.

3.1.1 Actuation and High Level Queries

Let's focus firstly on Actuation Queries. As said before, they allow the user to be able to set a certain number of parameters on a sensor (notice that a parameter can be a command, used, for example, to start an actuator installed on a node, to turn off power supply and even to upgrade the node firmware). The High Level Queries are instead used to define data structures that can be used both to contain the final results of a query, and as intermediary structures for other following manipulations. In PerLa two data structures have been defined:

- STREAM: it's the common type of data structure and can be seen as an unbound table that collects records produced by LLQs or by others HLQs (especially when this data structure is used as intermediary, as mentioned before);
- SNAPSHOT: it's a data structure representing a set of records produced in a given period and it is mainly used to implement *pilot join* operations which will be presented in the next subsection.

3.1.2 Low Level Queries

The statements composing the LLQs allow to precisely define the behaviour of a single device, and their main goal is the definition of sampling operations and the application of some SQL operators on sampled data (e.g. aggregation, grouping and filtering). Their role within PerLa query is fundamental and every LLQ is composed of at most four sections:

Sampling section

This part specifies how and when the sampling operation should be performed. The syntax allows to specify both a *time based* sampling (i.e. at a certain frequency) and an *event based* sampling (e.g. data is sampled only when an event occurs, such as the presence of a RFID sensor in an established area). SAMPLING is the operator used to accomplish these functionalities: it can be followed by an events list whose verification is used as a condition to execute sampling (event based sampling), or by a IF-EVERY-ELSE statements block which allows to define a precise sampling frequency upon the verification of some conditions (time based sampling, see Subsection 3.1.3 for an example). The REFRESH clause allows the user to specify how often the sampling frequency must be evaluated again during the lifetime of the query, and can however be used only in time based sampling. Independently from the chosen sampling type, the sampled values are appended into a *local buffer,* until they are manipulated by the data management section, described in the following.

Data management section

As mentioned before, this section has the role of managing sampled data in order to compute query results. This goal is achieved through the use of the SELECT clause, which is performed upon one or more records currently present in the local buffer: the computed records are then appended to an *output buffer,* ready to be sent to upper levels. In addition to SELECT clause, standard SQL aggregation operators are part of this section: AVG, MIN, MAX, COUNT, GROUP BY and HAVING clauses keep partially their SQL semantics [2], and allow the user to specify how to manage the sampled data when the selection is performed. Again, in Subsection 3.1.3 an example of selection and aggregation is given.

Execution condition section

This section defines the rules to establish if a certain sensor (or a group of sensors) should participate to query computation. Currently two clauses have been defined to accomplish this goal: EXECUTE IF and the aforementioned PILOT JOIN clauses. Such clauses are evaluated before the sampling section is executed, avoiding sensors (not involved in query execution) to waste resources: the sampling activity, in fact, always requires power and this could represent a problem on battery powered sensors. A dedicated component (see also Section 3.3) is employed to choose which sensors will be taken into account to compute query results. One final note is relative to the pilot join operation: it allows to define *context-awareness* and *data tailoring* capabilities in PerLa and more information about this topic can be found in [16].

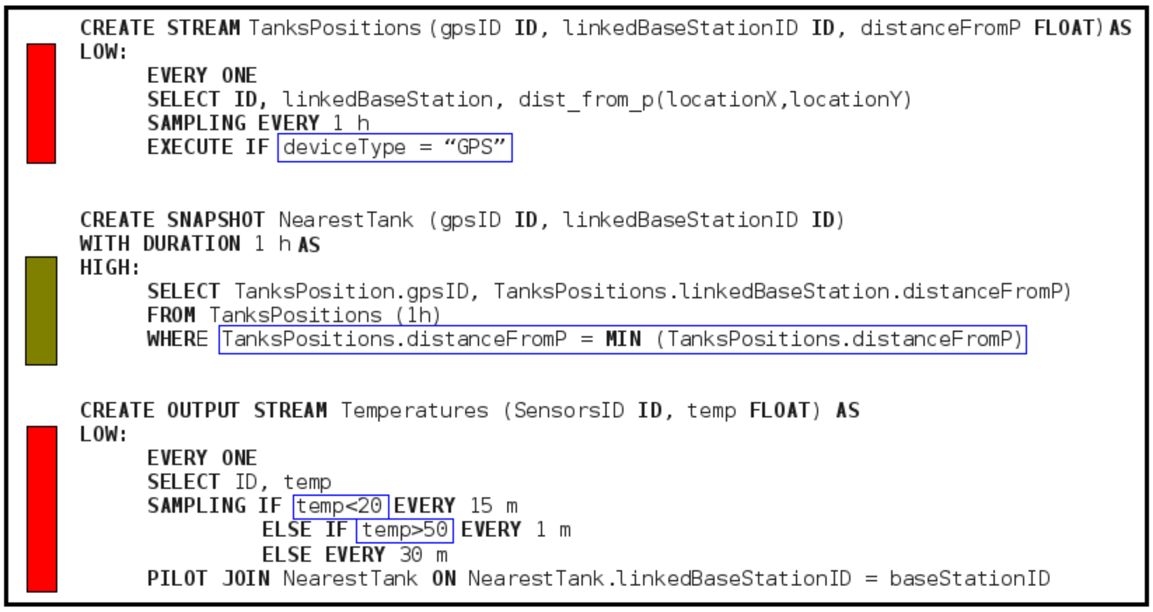
Termination condition section

The termination condition is the last block of an LLQ and allows the user to stop the query computation when a certain amount of time is elapsed or a given number of sampled values is collected. The TERMINATE AFTER clause is charged to accomplish such objective.

Complete EBNF of all statements presented here can be found in [2].

3.1.3 A PerLa query example

Having achieved the comprehension of language syntax and semantics, a quite complex query example related to a wine transport scenario is presented here. The query is used to keep temperature values under control, using sensors placed into the transport truck that is currently nearest to a given point P.



```
CREATE STREAM TanksPositions (gpsID ID, linkedBaseStationID ID, distanceFromP FLOAT) AS
LOW:
      EVERY ONE
      SELECT ID, linkedBaseStation, dist_from_p(locationX,locationY)
      SAMPLING EVERY 1 h
      EXECUTE IF deviceType = "GPS"

CREATE SNAPSHOT NearestTank (gpsID ID, linkedBaseStationID ID)
WITH DURATION 1 h AS
HIGH:
      SELECT TanksPosition.gpsID, TanksPositions.linkedBaseStation.distanceFromP)
      FROM TanksPositions (1h)
      WHERE TanksPositions.distanceFromP = MIN (TanksPositions.distanceFromP)

CREATE OUTPUT STREAM Temperatures (SensorsID ID, temp FLOAT) AS
LOW:
      EVERY ONE
      SELECT ID, temp
      SAMPLING IF temp<20 EVERY 15 m
               ELSE IF temp>50 EVERY 1 m
               ELSE EVERY 30 m
      PILOT JOIN NearestTank ON NearestTank.linkedBaseStationID = baseStationID
```

Figure 3.2: Query (quite complex) example, taken from [2]

In this example LLQ and HLQ are highlighted with a red and a green rectangle respectively. The reader can find all the PerLa keywords in **bold**, while a blue rectangle highlights PerLa statements which requires a physical computation to be analyzed: such elements are called *expressions*. Every expression can be seen as a set of constant elements (number, attributes to be sampled, etc) called *Constants* and a group of operators (addition, division, AND, OR, XOR, and many more) called *Nodes* [17].

3.2 The high level interface (middleware)

PerLa language needs to be fully supported by the *middleware*: a previous work within the ART DECO project [18] studied a possible design (later chosen as effective) which proposes a middleware architecture composed of three levels. The first level is the *application level*, whose main purpose is to manage user interaction by receiving queries to be executed. A *logical level* is inserted down below, in order to abstract from the underlying set of sensors. Finally a *physical level* is used to send and receive data from sensors.

3.2.1 Application Level

This level is the front-end used by applications in order to access data coming from the physical devices. It's composed of a *parser* [19], whose main purpose is to transform the user query (written in text format) to another internal representation, more suitable to be managed within PerLa middleware. This representation consists in a set of Java classes [20]: the parser separates the three aforementioned queries types (AQ, HLQ and LLQ), then a proper Java class is created for each SQL clause. The parser is also able to recognize also expressions, giving a similar Java representation used for SQL clauses [17], distinguishing between nodes and constants. The parser signals potential syntactic errors and the query is ready to be pushed to the next level when the parsing operation is successfully concluded.

3.2.2 Logical Level

The logical level represents the core of the middleware. Such fundamental role derives from the fact that this level contains a series of entities called *logical objects*, representing the heart of the whole PerLa architecture. The main idea behind logical object concept is to wrap sets of homogeneous sensors into a physical device abstraction called, indeed, *logical object* (see Figure 3.3). As it can be seen from the figure, logical objects allow interaction with physical devices and the middleware through the definition of a suitable interface, that provides three main functionalities:

- retrieving attributes
- firing notification events
- getting the list of supported attributes and events

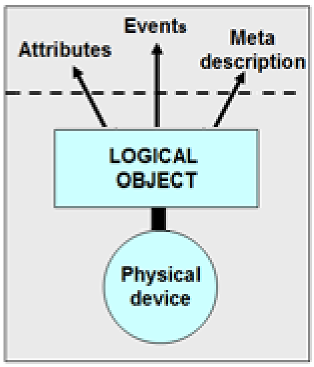


Figure 3.3: Logical object abstraction

It is important to underline that with the introduction of the logical object concept, heterogeneity management challenge is achieved, since pervasive system can now be abstracted as a *group* of logical objects also hiding complexity to the end users.

It's now possible to go deeper into logical objects, to better comprehend how its three functionalities can be realized. In fact the logical objects implementation includes two important components: a *Low Level Query Environment*, containing all the data structures and algorithms needed to compute LLQ result, and a *Functionality Proxy Component* (shortened *FPC*), charged of exchanging data with physical devices.

Low Level Query Environment

LLQ Environment is going to contain the real "engine" of LLQ execution as well as the structures needed to accomplish such goal. This component is introduced here to highlight its role within logical object and its relation with FPC. However more details about this component can be found in following chapters since this thesis focuses on this important part.

Functionality Proxy Component (FPC)

FPC is the object that really implements physical device abstraction within a logical object. FPC is also the object that is charged to manage devices initialization and their addition to the system. Such goal is achieved through a device self-description: device type, capabilities, attributes that can be sampled and many other information are presented to the system using a XML file. First studies about FPC component, together with more details about the XML descriptor and the adopted communication and data exchange protocols, are reported in [7], while a partial implementation and integration within PerLa is explained in [8].

3.2.3 Device access (physical) Level

This level is mainly composed of a *C* library: it should be noticed that most of nowadays sensors aren't capable of running Java code (and no code at all in the case of RFID sensors tags). Such library has been designed with the purpose of minimizing the low level programming effort required to the developer user to integrate a new technology in the middleware: when programmers need to integrate a new device into PerLa, they just need to extend and recompile the library and to define an XML description file, in order to make device and FPC dialogue possible. More information on library architecture can be found in [6].

As important conclusion, it can be noticed that the logical level exposes interfaces on one hand to a Java "world" (that's to say, the application level), and on the other hand to a *C* "world" (the device access level, through the FPC) making possible all the objectives prefixed in the introduction of the chapter.

3.3 First PerLa deployment: rockfall prevention

Having introduced both the language and the middleware features it's now possible to give an overview on PerLa functioning, analyzing how these two aspects interact between themselves. Let's suppose that a query (indicated with a red arrow in Figure 3.4) is injected in PerLa.

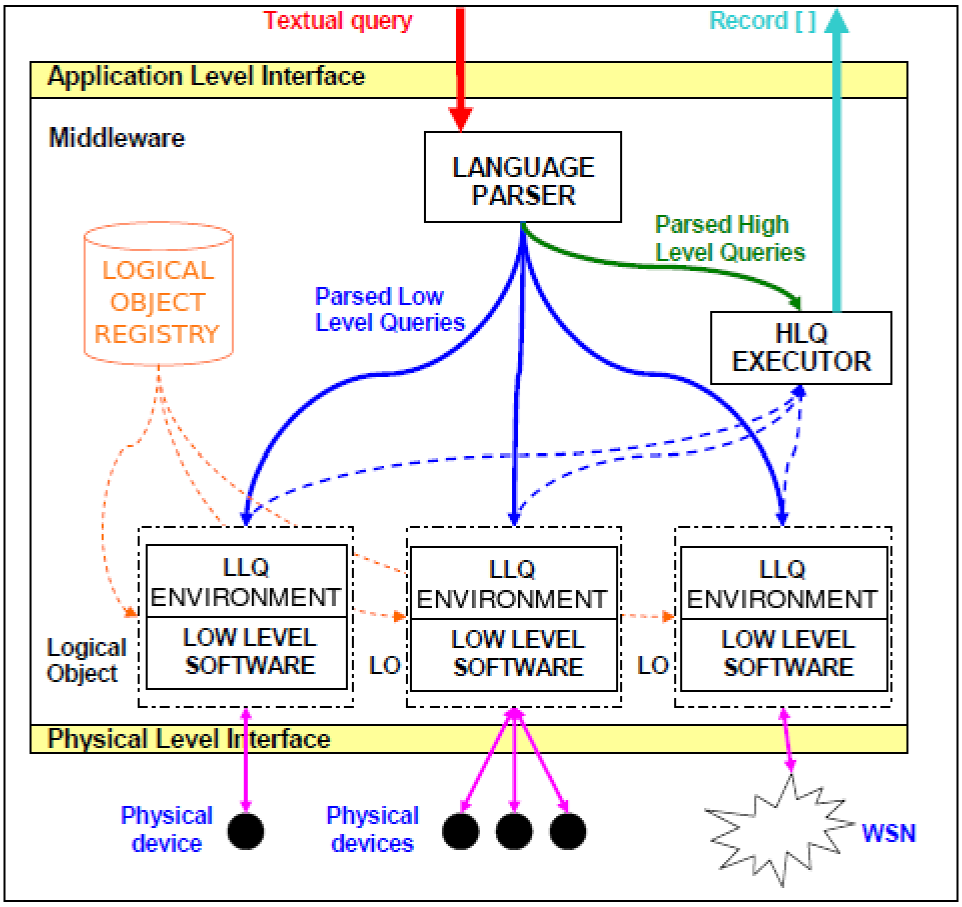


Figure 3.4: PerLa middleware overview

As a preliminary passage, the textual query is analyzed by the parser obtaining the Java representation explained in Section 3.2.2. LLQs obtained from parser can now be injected in the proper logical objects using a dedicated component called *Logical Object Registry* (or simply *Registry*); it contains references to logical objects currently active in the system, allowing the LLQs injection process to work correctly. Query sampling and data management sections (blue arrows) are then activated: thanks to the combined action of the LLQ Environment components and *Low Level Software* (that's in fact the name of FPC and *C* library considered together) the data can be received, computed and grouped again into an *HLQ Executor,* where the query final result is computed and presented to the user (hatched blue and light blue arrows).

It's now worth to briefly analyze where, within a pervasive system, every single component of a PerLa query is physically executed. The parsing of the query, as well as the evaluation of the set of logical objects to be taken into account for query computation (evaluating the EXECUTE IF clause), are executed by the highest part of the middleware, that is typically deployed on the server machine used to monitor the pervasive system. The high level execution is, at the same way, executed on the same machine, since it manipulates data streams originated by LLQs (but a distributed version of the HLQ engine can probably be designed). The location where a LLQ execution is performed is, indeed, a more delicate issue and the computation capabilities of the nodes composing the pervasive system must be taken into account. If node capabilities allow the node itself to run a Java virtual machine, both Low Level Software (i.e.: FPC) and Low Level Query Environment can be deployed on the node. This is the case of powerful devices, such as netbooks, PDAs or ad-hoc boards, but this is not the most general case. As an example, consider the pervasive system that will be employed in [3], where small sensors called *Acquisition and Elaboration Units* (shortened *AEUs*) will be used to monitor and prevent rockfalls. As can be seen from Figure 3.5, these small devices are linked together using *CAN-bus*, while a *Local Coordinator* acts as a master on the bus, sending collected data to a gateway through a *ZigBee* network. The single AEU, as well as every Local Coordinator, are not powerful enough to run a Java virtual machine: Low Level Execution must than be achieved using the nearest device (connected to the middleware) that is able to host the Java virtual machine. In Figure 3.5 such device is called *Gateway* and it hosts the Low Level Software as well as the Low Level Query Environment. As mentioned before, the High Level Execution is finally achieved on the machine used to monitor the pervasive system, connected to the Gateway using a 5 GHz *TCP/IP* radio bridge.

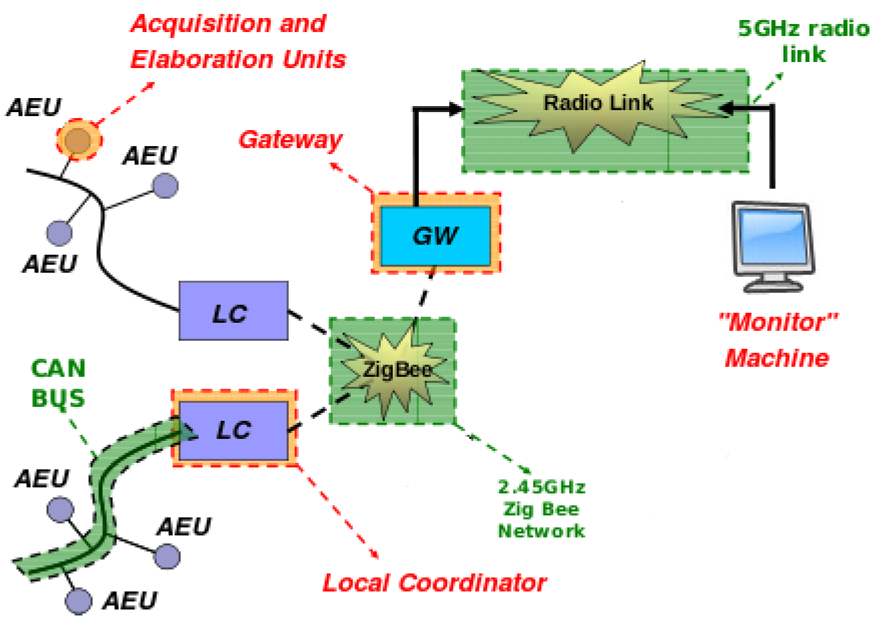


Figure 3.5: An example of a pervasive system [3]

As conclusion, it must be noticed that the situation presented is very common in most of pervasive systems; thus, as long as the nodes aren't able to compute Low Level Query Execution, a fully capable machine needs to be introduced in order to supply such lack of computational capabilities. If, instead, a node is “intelligent” enough to execute the LLQ, the computation is endorsed by the node itself. Finally notice, as an extreme case, that nodes could only partially support the execution of a LLQ, executing only a subset of its four sections: again, the nearest capable device will supply for the sections that can't be directly executed on the node.

4 LLQ Execution Design

This chapter is focused on the design phase of the Low Level Query Execution. The dissertation will highlight the need of the Low Level Query Execution within PerLa, finally describing the design choices adopted.

4.1 The importance of LLQ Execution

Before entering into LLQ Execution details it's worth to analyze this important aspect. As said in previous chapters, the FPC component must be placed on the most capable device nearest to the pervasive system since it's written in Java and thus requires a Java Virtual Machine to operate. Since FPC is strictly linked with the device, delegating the LLQ Execution to the FPC component might sound as the better choice to achieve LLQ Execution. During FPC first studies, this component was always designed as the key component "only" charged of masking the complexity of the pervasive system to the upper level, following the well known *divide-et-impera* software engineering approach. This is why FPC tasks don't include data collection mechanisms nor data processing capabilities. This fact brings us to the important conclusion that one (or more) PerLa component(s) must be employed to fulfill LLQ Execution. The following example enforces this important statement, and is exposed in Figure 4.1

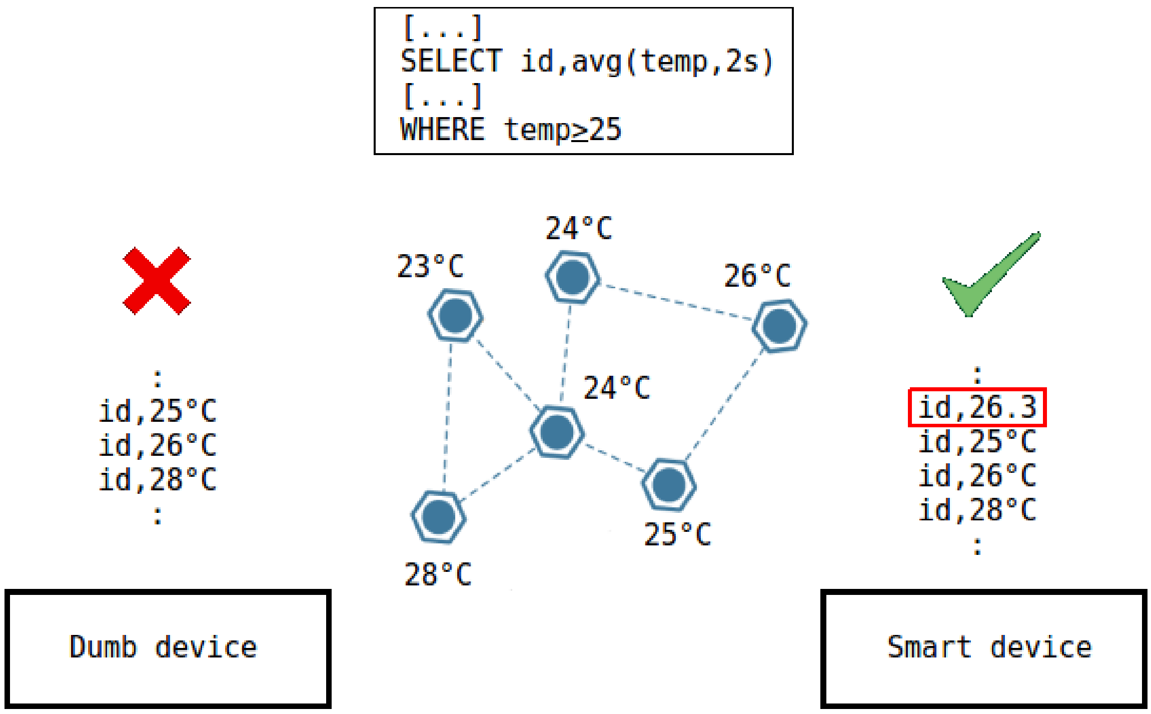


Figure 4.1: Example with "dumb" and "smart" devices

In the previous figure an hypothetical PerLa instance is used to monitor a sensor network composed of simple devices able to sample only temperature, using the query depicted in the same figure. On the sides of the figure we find two scenarios: on the right we suppose that the devices composing the network are fully capable devices, i.e. capable to manipulate data, such as calculating a particular mathematical functions as well as aggregates (AVG,MAX,MIN,COUNT etc). Such devices are called "smart". On the left side, indeed, we suppose that our devices are only able to sample data and are completely deprived of any computational power. We call these devices "dumb". Let's now focus on the example. The query requires to compute an average value to be successfully completed, but, as said before, FPC tasks don't include data processing. This combination yields an important conclusion: if the device is not powerful by means of computational power, the query **cannot** be executed. This is why LLQ Execution is so important and necessary within PerLa: it supplies the computational lacks of the “dumb” sensor. This fact is even more important since most of nowadays sensors have not (or have partially) the characteristics that make them a “smart” sensor.

4.2 LLQ Execution design

Now that the importance of LLQ execution has been clarified, the LLQ Execution design can be presented. As mentioned in the previous chapters it can be summarized, from a general point of view, as composed of two important parts:

- *Data collection (from the pervasive system);*
- *Data processing and result computation.*

Each part requires a complete design process whose details are presented in the following sections. Before entering into such details it is possible to analyze how these two parts cooperate themselves to achieve LLQ Execution, following the usual topdown approach. Since it's the first to be performed, let's firstly focus on how to obtain data, executing the sampling section of an LLQ. Collecting data means, in PerLa, to fill a *local buffer* with a certain amount of data retrieved from the pervasive system (precise number however differs from query to query). Design phase started studying how this process could be performed, introducing the idea of the *DataCollector* entity, that is primarily charged, as suggested by the name, to the collection and filtering (WHERE clause) of values from the pervasive system. Once the local buffer is filled with values the data management section of the LLQ can compute results. This task is completed by another entity called *LLQ Executor* that withdraws the local buffer values, process them and append the final results to an *output buffer*. This is obviously the most general view of the LLQ Execution and is highlighted in the following figure.

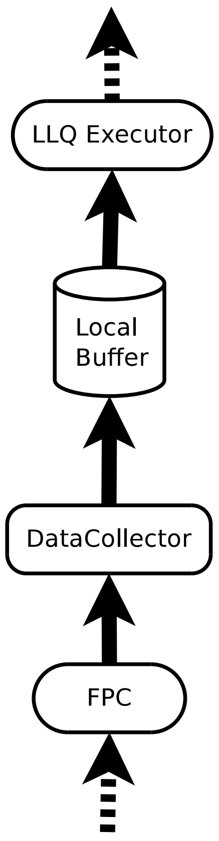


Figure 4.2: Entities designed to achieve LLQ Execution

Hereafter a complete example of how LLQ Execution is supposed to operate, as well as what has just been discussed, is presented. Let's suppose that PerLa is operating on a pervasive system composed of four nodes, equipped with the following on-board sensors:

- **Node A:** temperature, pressure;
- **Node B:** temperature, pressure, humidity;
- **Node C:** humidity;
- **Node D:** brightness.

Let's initially suppose that only one query, reported in 4.1, is injected into the system.

Algorithm 4.1 First query

```
CREATE STREAM TempNodeA (TIMESTAMP TS, FLOAT temp) AS
LOW:
        EVERY 20 m
        SELECT TS, AVG(temp,20 m)
        SAMPLING EVERY 10 m
        EXECUTE IF node_id = "Node␣A"
```

After the query analysis conducted by the parser, the *Logical Object Registry* is used to choose which FPCs will be involved into the query computation, evaluating the EXECUTE IF clause presented in previous subsections. In this example the registry will contain references to four FPCs (i.e. **FPC A**, **FPC B**, **FPC C** and **FPC D**), corresponding to the four nodes presented above. Consulting the registry it is possible to discover that only **FPC A** satisfies the EXECUTE IF clause of the query. At this point the computation will be completed in the following steps:

1. Every 10 minutes a temperature value is sampled (according to the SAMPLING clause) by **FPC A** and inserted to a proper *local buffer,* along with proper timestamp and ID values;

2. Every 20 minutes the average value (reported in the SELECT clause) is computed and inserted into the output stream *TempNodeA*. Notice that streams (and, more in general, also snapshots) endorse the role of *output buffer* mentioned in Subsection 3.1.2;

Note that points 1 and 2 respectively correspond to *DataCollector* and *Low Level Executor* entities, and they allow the query to be computed correctly. This situation is reported in Figure 4.3.

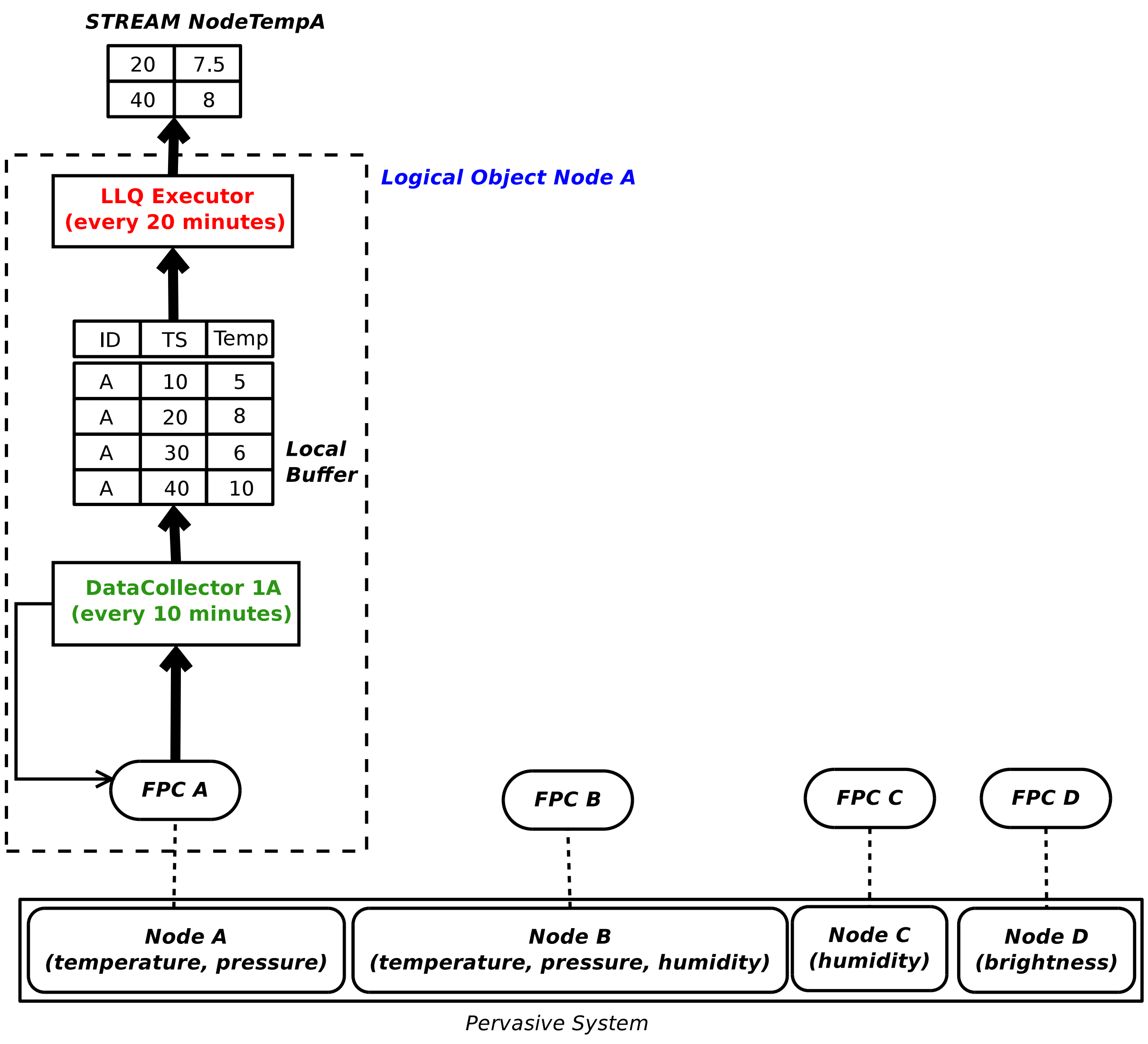


Figure 4.3: LLQ Execution (single query)

This example has been focused on a single query running into the system: in order to explain how LLQ Execution deals with multiple queries running simultaneously, let's suppose that another query, hereafter reported, is injected into the same system of the previous example.

Algorithm 4.2 Second query

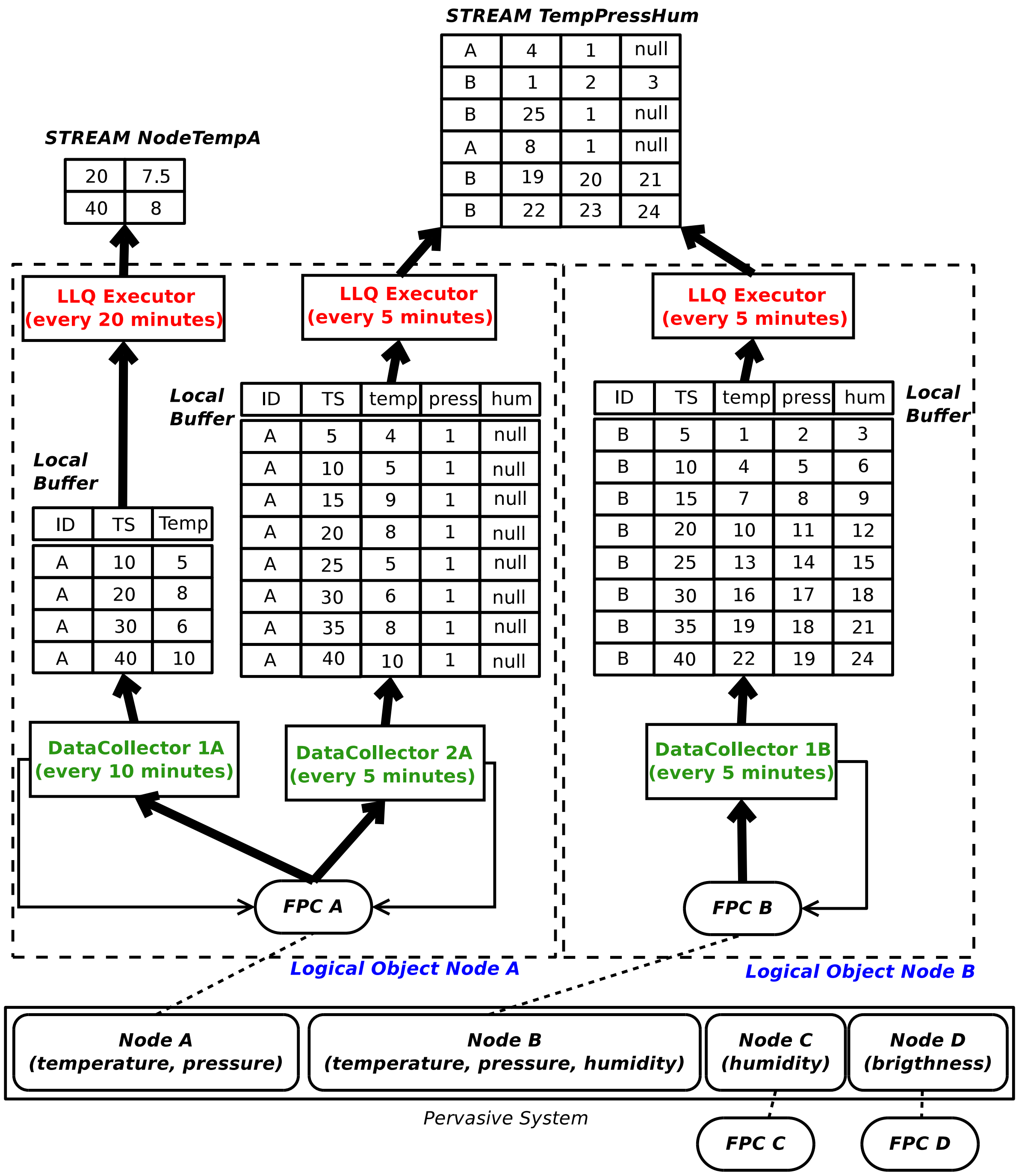
```
CREATE STREAM TempPresHum (TIMESTAMP TS, FLOAT temp, FLOAT pres, FLOAT hum) AS
LOW:
        EVERY 5 m
        SELECT TS,temp,pres,hum
        SAMPLING EVERY 5 m
        EXECUTE IF EXISTS(pres)
```

As said for the first query, the *Local Object Registry* is used to discover which FPCs will be involved: since this second query requires to be executed on all the nodes having a pressure sensor on-board, the registry returns both **FPC A** and **FPC B** as result. The computation of this query then requires the following steps to be completed:

1. Every 5 minutes, temperature, pressure and humidity values are sampled through both **FPC A** and **FPC B**, and inserted in *two* distinct local buffers. Notice that **FPC A** doesn't board the humidity sensor but it is anyway included into query computation since, from the point of view of the *Local Object Registry*, it boards the required pressure sensor. The values inserted by **FPC A** into its proper local buffer will then contain a *null* value in the humidity field.

2. Every 5 minutes the data management part is activated on each of two local buffers and results are inserted into the final stream *TempPresHum*.

This situation, that includes two queries running simultaneously, is reported in Figure 4.4 and highlights some important aspects of LLQ Execution, explained after the figure.


STREAM TempPressHum
STREAM NodeTempA
LLQ Executor (every 20 minutes)
LLQ Executor (every 5 minutes)
LLQ Executor (every 5 minutes)
Local Buffer
Local Buffer
Local Buffer
DataCollector 1A (every 10 minutes)
DataCollector 2A (every 5 minutes)
DataCollector 1B (every 5 minutes)
FPC A
FPC B
Logical Object Node A
Logical Object Node B
Node A (temperature, pressure)
Node B (temperature, pressure, humidity)
Node C (humidity)
Node D (brigthness)
Pervasive System
FPC C
FPC D

STREAM NodeTempA

20	7.5
40	8

STREAM TempPressHum

A	4	1	null
B	1	2	3
B	25	1	null
A	8	1	null
B	19	20	21
B	22	23	24

Local Buffer

ID	TS	Temp
A	10	5
A	20	8
A	30	6
A	40	10

Local Buffer

ID	TS	temp	press	hum
A	5	4	1	null
A	10	5	1	null
A	15	9	1	null
A	20	8	1	null
A	25	5	1	null
A	30	6	1	null
A	35	8	1	null
A	40	10	1	null

Local Buffer

ID	TS	temp	press	hum
B	5	1	2	3
B	10	4	5	6
B	15	7	8	9
B	20	10	11	12
B	25	13	14	15
B	30	16	17	18
B	35	19	18	21
B	40	22	19	24

Figure 4.4: LLQ Execution example (two simultaneous queries)

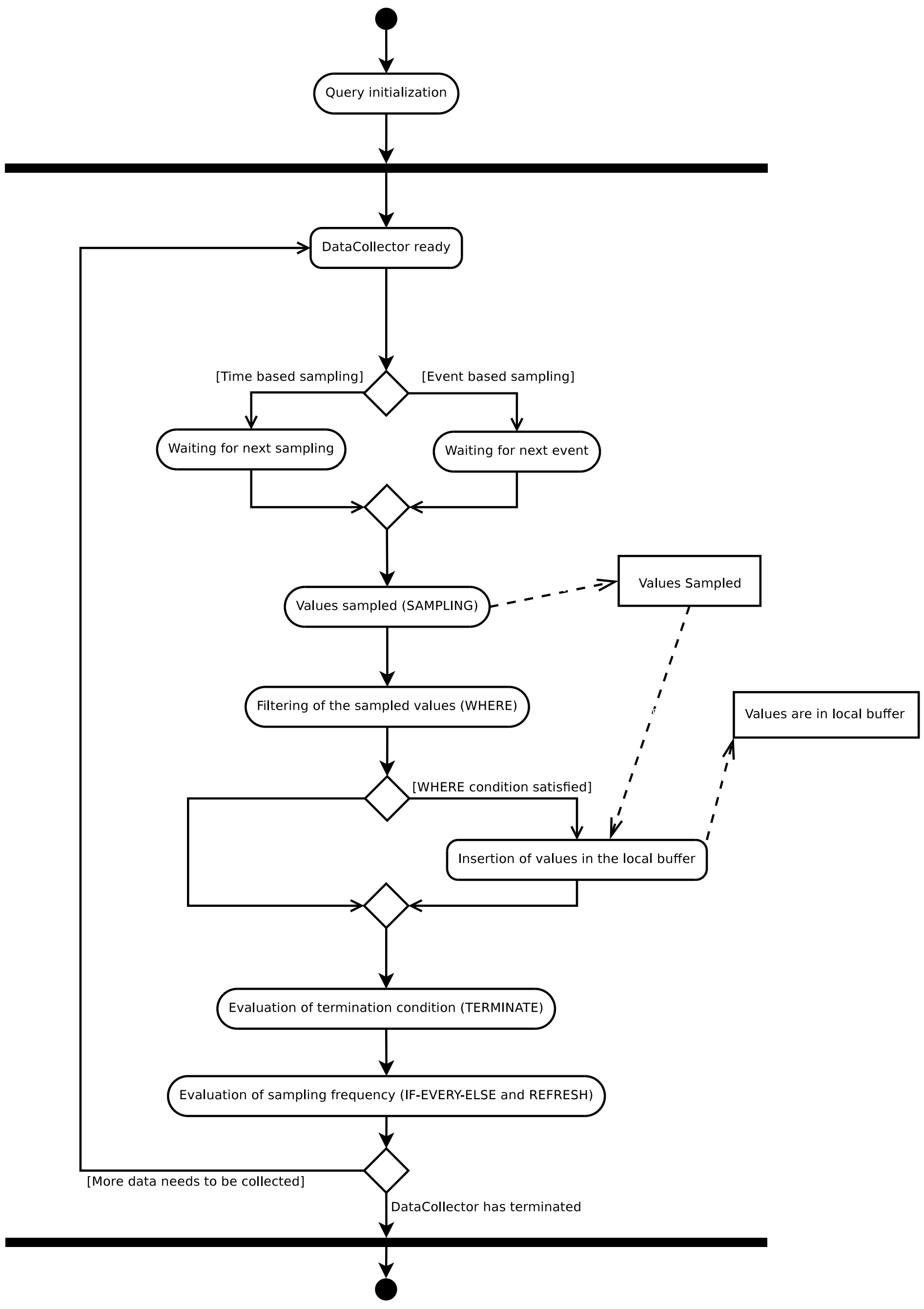
In fact, for each FPCs reported by the *Local Object Registry*, a proper DataCollector must be used; moreover, for each DataCollector, a proper local buffer, as well as a proper LLQ Executor, need to be employed to compute successfully the query result. This "rule" can be summarized saying that an instance of the tuple <*DataCollector, Local Buffer, LLQ Executor*> needs to be instantiated for each FPC involved into a LLQ computation. As conclusion, it can be noticed that a LLQ produces a flow of records potentially coming from different FPCs, but each of these records is necessarily the result of a computation made on a set of attributes sampled on the same FPC.

4.3 *DataCollector* entity design

It's now possible to focus the attention on the design of the DataCollector, starting from its lifecycle. DataCollector needs to be created and modeled on the LLQ that is going to serve: sampling method (time/event based), WHERE and termination condition, sampling frequencies are information that DataCollector must be informed about before it can start operating. DataCollector modeling is carried out at the beginning of its lifecycle during query initialization phase thanks to a specific PerLa component called *Query Analyzer*: it receives the Java representation of the query as it is produced by the parser and it creates every structure needed to compute the query (see also the following example). Among these structures an ad-hoc DataCollector is created for each query analyzed. QueryAnalyzer is, in fact, able to navigate through the received Java structures, searching for terminate conditions, sampling type (time/event based), REFRESH and WHERE clause conditions, and produces as output a DataCollector whose activity diagram is fitted for the query analyzed. Once DataCollector is created it can start operating: the following subsection focuses on this aspect.

4.3.1 DataCollector activity

Data collection activity is represented in Figure 4.5 using an UML activity diagram.



Query initialization
DataCollector ready
[Time based sampling]
[Event based sampling]
Waiting for next sampling
Waiting for next event
Values sampled (SAMPLING)
Values Sampled
Filtering of the sampled values (WHERE)
Values are in local buffer
[WHERE condition satisfied]
Insertion of values in the local buffer
Evaluation of termination condition (TERMINATE)
Evaluation of sampling frequency (IF-EVERY-ELSE and REFRESH)
[More data needs to be collected]
DataCollector has terminated

Figure 4.5: Data collection UML activity diagram

In this figure, the DataCollector creation discussed before is represented by the "query initialization" block. The next step is the execution of sampling section according to the sampling type:

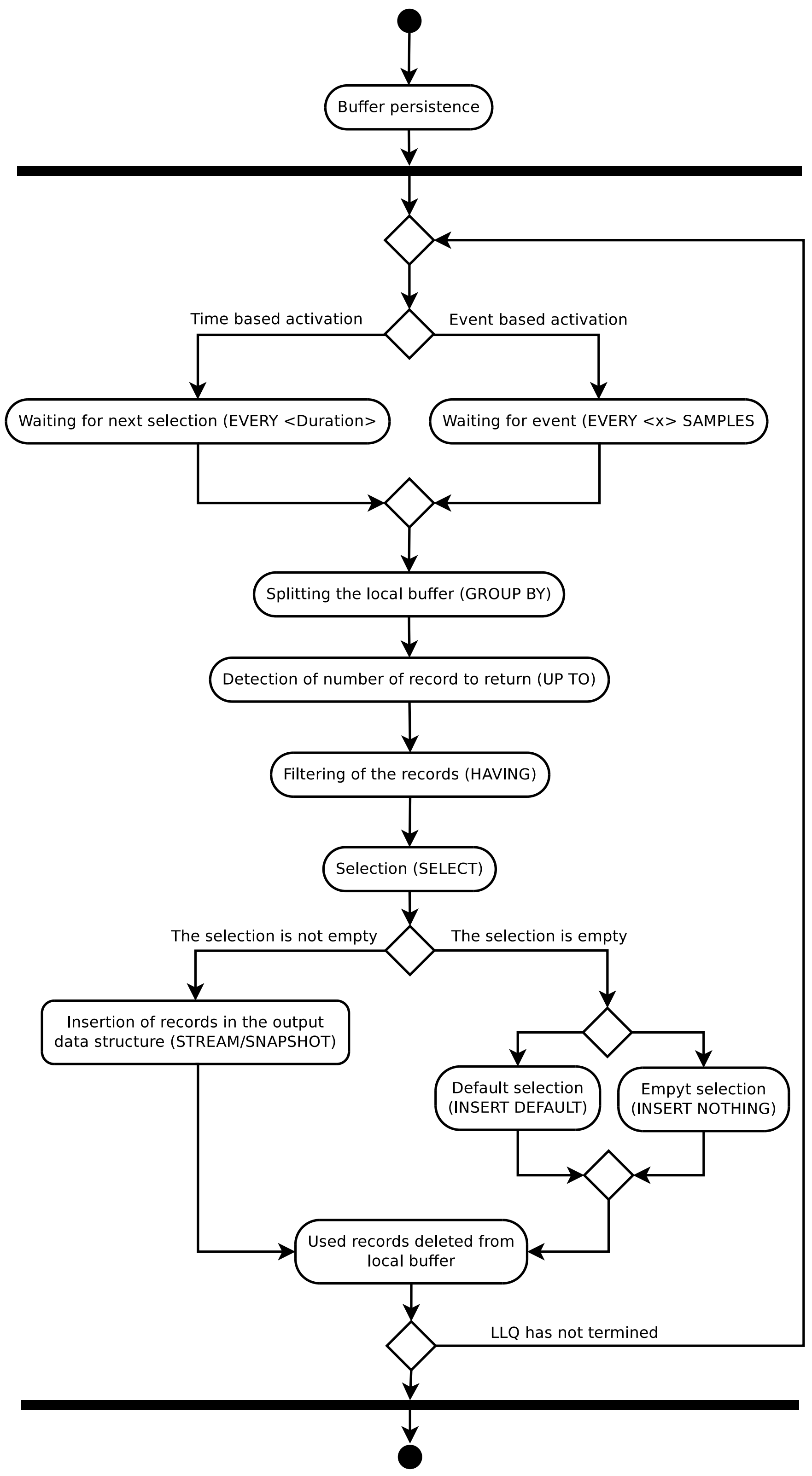
- Time based sampling: DataCollector waits for next sampled set of values according to sampling frequency. The IF-EVERY-ELSE block is used to decide the sampling frequency to adopt.

- Event based sampling: the event causing sampling is waited for until it occurs (ON EVENT clause).

Indifferently from the employed sampling type, a set of sampled attributes (retrieved from the pervasive system) is available when the sampling section execution is concluded. The WHERE clause, which is the only clause of data management part that data collection has to deal with, is then activated, allowing DataCollector to discard values according to a condition specified in the clause: only those records which satisfy the WHERE condition are inserted into the local buffer. DataCollector next step is the management of the REFRESH clause, in order to discover potential changes in the sampling frequency. The final DataCollector task is the evaluation of termination condition, to detect if data collection process has to be terminated. According to this condition, DataCollector is able to stop its activity; otherwise, a new set of values is waited for and the cycle shown in the Figure 4.5 is repeated again.

4.4 Data processing and result computation

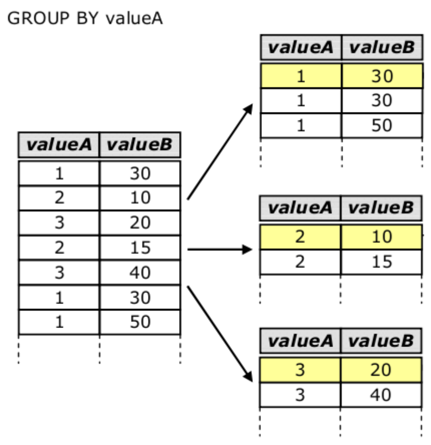
While DataCollector is operating, the sampled and filtered data are continuously appended into the local buffer and are ready to be transformed by the data processing mechanism into query results. As said in previous sections this task is fulfilled by the *LLQ Executor* component, whose activity is reported in the following UML diagram.



Buffer persistence
Time based activation
Event based activation
Waiting for next selection (EVERY <Duration>
Waiting for event (EVERY <x> SAMPLES
Splitting the local buffer (GROUP BY)
Detection of number of record to return (UP TO)
Filtering of the records (HAVING)
Selection (SELECT)
The selection is not empty
The selection is empty
Insertion of records in the output data structure (STREAM/SNAPSHOT)
Default selection (INSERT DEFAULT)
Empyt selection (INSERT NOTHING)
Used records deleted from local buffer
LLQ has not termined

Figure 4.6: LLQ Executor activity UML diagram

The first step is to make persistent the values that are within the local buffer. At this point, in fact, sampled values "live" in the volatile memory of the physical machine that is hosting the running PerLa instance. Persistence is realized by saving the sampled values in a SQL table of a chosen DBMS running on the same machine. Currently a dedicated project is dealing with persistence, and the interested reader can find more details in [6]. Once that data is made persistent, LLQ Executor evaluates if the selection to be performed is time or event based. Independently from the selection type detected, the LLQExecutor enters the core of its activity: the GROUP BY clause is evaluated producing the division of the local buffer according to the clause condition. An example of how the local buffer is split is reported in the following figure:



valueA	valueB
1	30
2	10
3	20
2	15
3	40
1	30
1	50

valueA	valueB
1	30
1	30
1	50

valueA	valueB
2	10
2	15

valueA	valueB
3	20
3	40

Figure 4.7: GROUP BY clause example, taken from [2]

After the evaluation of the GROUP BY clause, LLQ Executor evaluates the UP TO clause in order to discover if there's a maximum number of records to be inserted into the final structure (SELECT clause is identical to the SQL one, and the number of selected records is unknown when the clause is executed). Finally the HAVING clause acts as filter to discard data according to a particular condition. At this point the SELECT clause can be executed with two potential result types: if no record is produced then the user can choose to insert anyway a default record into the final data structure (INSERT DEFAULT) or to take no actions (INSERT NOTHING). The values contained in the local buffer that have been used in this selection are now unuseful and can be eliminated. Like DataCollector, the LLQ Executor cycle must be repeated again if LLQ has not been terminated. In the same way LLQExecutor is customized for the query that it's going to serve by the *Query Analyzer* component introduced before. LLQ Executor and DataCollector finally share the same end: when the LLQ is terminated they're eliminated.

5 FPC interface design

From the point of view of the flow of data, the collection of sampled values presented in the previous chapter represents the first step of a long chain that ends with the presentation of the final results to the final user that has injected the query. At this point of dissertation it should be clear that FPC is linked to three fundamental PerLa components:

1. The device that is abstracting and thus representing
2. The Logical Object Registry
3. The DataCollector entity

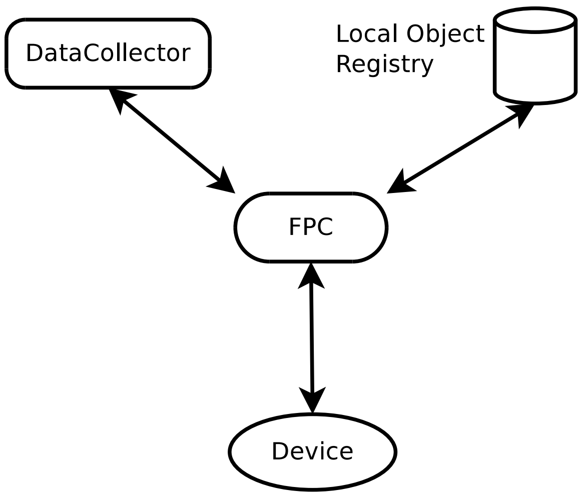


Figure 5.1: FPC relationships with other PerLa key components

A well-defined interface required to be designed towards every one of these entities, and is explained in the following subsections.

5.1 Interface towards DataCollector

Let's then focus on the interface used by FPC to communicate with the DataCollector. As mentioned in Section 4.2 DataCollector lifecycle begins thanks to the activity of the Query Analyzer component. Since for each FPC involved into query computation an instance of the tuple <*DataCollector, Local Buffer, LLQ Executor*> needs to be created, it requires the FPC interface to expose the following capabilities:

- the capability of creating a dedicated channel, where sampled data can flow within, starting from the FPC and ending with the DataCollector.
- the capability of allow forced sampling. Such capability is required by the REFRESH clause, to retrieve a group of records (or a single one) in order to evaluate a new sampling frequency.

The Section 6.2.1 will explain what kind of problems such capabilities imply.

5.2 Interface towards the Logical Object Registry

The relationships between the FPC and the Logical Object Registry can be divided into two separate sides: one, when every FPCs is created and another when a particular FPC is chosen (by the Local Object Registry) to collaborate in computing query results. The dissertation is then divided for each one of such phases.

5.2.1 Interface exposing FPC attributes

Since the FPC represents the abstraction of every device type in the WSN, the Local Object Registry (LOR hereafter) needs to be informed about what kind of device the FPC is currently representing. This aspect is very important, since the execution condition section of an LLQ might require to query the registry, aiming, for example, at finding a particular device which responds to particular characteristics (EXECUTE IF clause). The reader might state that this goal could be satisfied when PerLa firstly starts up, simply polling the present FPCs. It must instead be remembered that PerLa supports heterogeneity with a plug&play fashion: if the user, at instant t_0, plugs in a new device that requires a new FPC not present in the system at t_0, this FPC needs to be created and added to the system. A series of information need then to be communicated to the LOR by the FPC just created and added. This flow of information requires the FPC to implement the capability of communicate what device is representing, by means of communicating how attribute can be sampled by the abstracted device. All the queries that will be injected at instants $t > t_0$ will then be able to take advantage of the device added.

5.2.2 Interface for query injection

When one (or more) FPC is selected into the set of FPCs needed to compute the result of a specific query, the LLQ must be registered on the FPC. In other words, FPC needs to expose the capability to "receive" and keep an internal trace of the LLQs at whose computation is collaborating. Since is the LOR that discriminate which FPCs are meant to execute an LLQ, this part of the FPC interface must be composed of two capabilities:

- The capability to allow the LOR to "register" the LLQ on the FPC and to signal it to start computation (i.e. the collection of data and the execution of the data management part of the LLQ)
- The capability of "unregister" the LLQ from the FPC, according to the conditions present in the termination condition section of an LLQ (if present)

5.3 Interface towards the abstracted device

As said in Subsection 3.3 the FPC is the component which also deals with the device level of the middleware. For this important task the FPC interface must provide the following capabilities:

- The capability to discover the data format adopted by the device (Little/Big Endian, number of bits adopted to represent a data type, etc) and to convert it into the format adopted within PerLa
- The capability to establish a channel between the FPC and the abstracted device: this capability was designed and partially implemented in [8], but a precise interface needed to be designed.

6 LLQ Execution Implementation

In this chapter the implementation of the design aspects proposed in the previous chapter is explained. Until now, terms like “query”, “sampled values” and “structure” have been used in a general fashion and always referring to the functionalities of these components. Section 6.1 is intended to explain what these components actually are, and which data structures are involved.

6.1 Data structures implementation

The first data structures that are examined in the following are those charged of representing sampled values generated by FPC.

6.1.1 Data structures for records

Let’s concentrate on sampled values: they can be grouped in *records*, each of them composed of a set of fields representing a sampled value. A suitable record structure must be defined, starting from the internal structure of each field, in order to allow an efficient access to the contained data. The classes presented here have been designed trying to accomplish this purpose.

QueryFieldStructure class

This class contains information about the internal structure of a record field

```
private Class<?> pFieldType;

private String pFieldName;

private int pFieldIndex;
```

The first variable represents the type of the value contained in the field; the second one represents the name of the variable contained in the field, while the third is the index number of the field within the record. As an example, suppose that a field is used to represent a temperature. In this case the three variables will assume the following values: "ConstantFloat.class" (a Constant to represent float values [17]), "temperature", and "0" (since we're talking about a single field in a record). Some methods (setter/getter) have been added to the class, allowing to read and write the variables.

RecordStructure class

This class defines the precise structure of a record and maps the name and position of each field into the record. Mapping is accomplished using HashMap built-in Java class, as shown.

```
private HashMap<String, QueryFieldStructure> pRecordStructure;
```

The first argument of the HashMap is the name of the attribute (and so of the field) that has been sampled (e.g. "temperature"). The second argument is the reference to the field structure whose name is given as the first argument. An internal method has been implemented in order to retrieve the index of a field given its name (this is very useful, for example, when we want to obtain the index of the "timestamp" field, if present). A similar method has been implemented in order to retrieve a field given its index.

Record class

This class is deputed to physically store sampled data. Its state is composed of

```
private Constant[] pConstant;

private QueryRecordStructure pQueryRecordStructure;
```

The array contains the constants (i.e., the actual sampled values), while the second variable is the record structure. Methods to retrieve record contents have been implemented as shown in the UML diagram in Figure 6.1. This figure is a summary diagram representing the interaction among the three classes presented before.

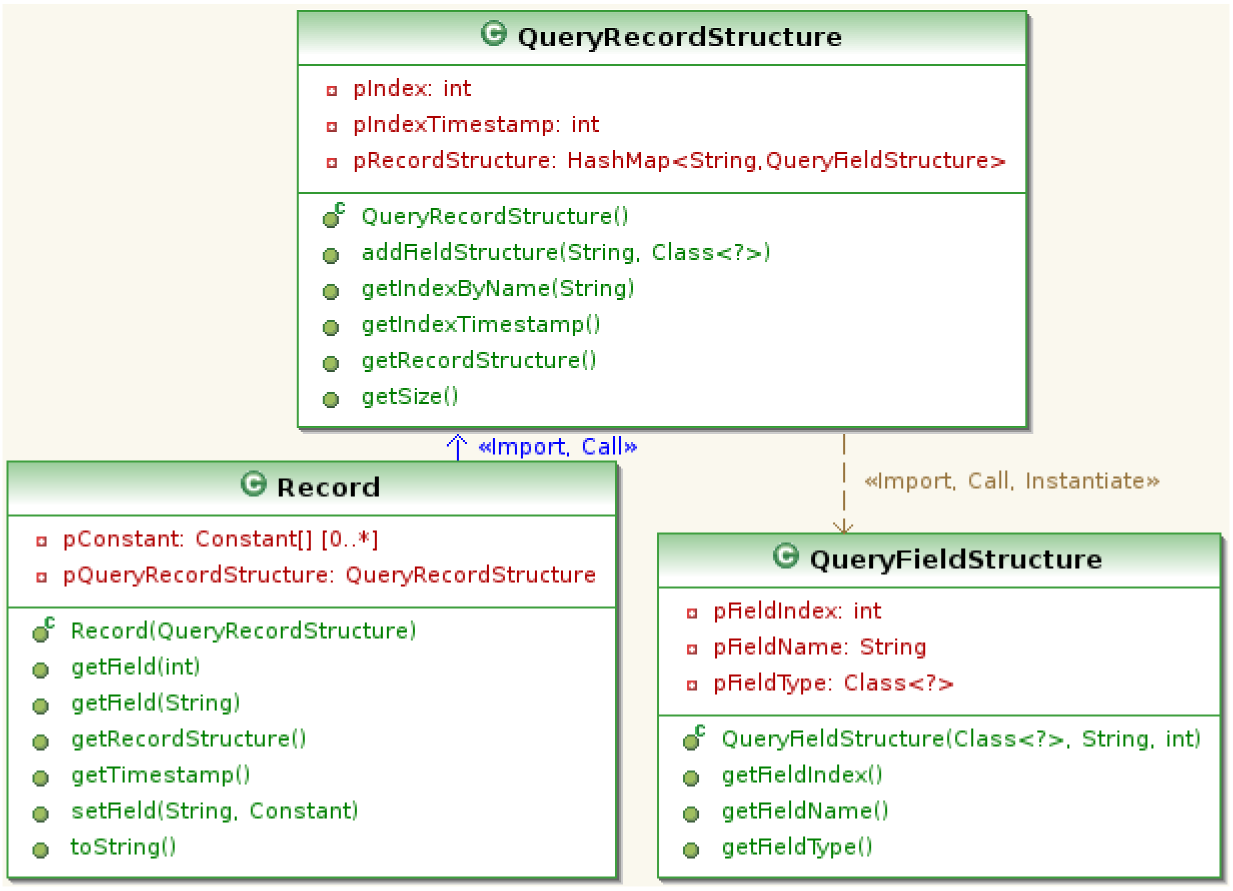


Figure 6.1: Data structures for records UML diagrams

NodeLogicalObjectAttribute class

NodeLogicalObjectAttribute [17] is the object charged of representing attributes (always contained in expressions) of a query, and it has been designed and implemented only at a mockup level. A more detailed implementation has been achieved in this project. When a textual query is injected, the parser creates, for every attribute found, an instance of this class filling two variables:

```
private String pIdentifier;

private Constant pValue;
```

Only the attribute name is initially filled since is the only information known at parsing time (pValue still needs to be sampled and its value is unknown). The second variable will be filled only at LLQ execution time, when the value will be physically retrieved. This class exposes classic getter/setter methods, that are used during expression evaluation by an ad-hoc class explained in the Subsection 6.1.3.

6.1.2 Data structures for query computation

Records presented in the above section must be inserted into suitable structures before being computed (in the case of local buffer) or sent to the upper level (in the case of output buffer). The Buffer class is used to achieve this goal.

Buffer class

Even if local and output buffers have different roles, their functionalities can be accomplished by the same class. The storing function is realized using Java ArrayList class which allows dynamic insertion and deletion, automatically fitting the size of the buffer whenever a record is appended or removed:

```
private ArrayList<Record> pRecordSet;
```

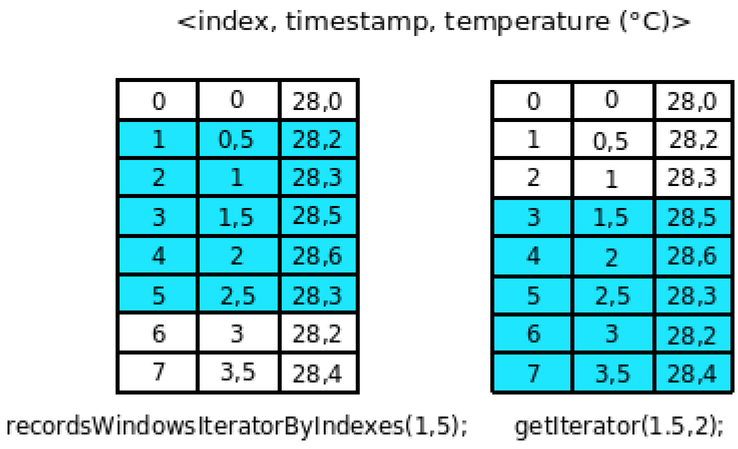
The real importance of buffer class lies in the methods that are provided to navigate through a set of records. The following methods are defined

```
public Iterator<Record> recordsWindowIteratorByIndexes(int parInit, int parEnd)
```

```
public Iterator<Record> getIterator(Timestamp parTimestamp, long parDelta)
```

The first one allows to obtain a set of records (from `parInit` index and ending with `parEnd` index) and, more important, to navigate through it using the iterator functionalities offered by Java. This method is particularly useful when computation of query results needs more than one record to be completed.

The second method is similar to the previous one, but different parameters types are used to iterate on the buffer: a starting timestamp is given as the start point and a set of records, ending at timestamp `parTimestamp + parDelta`, is returned. This method is useful when computation requires to operate on a group of records that have been sampled within a certain interval (e.g AVG(TEMPERATURE,30 S)). Figure 6.2 shows the behaviour of these two methods on a buffer taken as example, coloring in light blue the set of records returned.


<index, timestamp, temperature (°C)>

0	0	28,0
1	0,5	28,2
2	1	28,3
3	1,5	28,5
4	2	28,6
5	2,5	28,3
6	3	28,2
7	3,5	28,4

recordsWindowsIteratorByIndexes(1,5);

0	0	28,0
1	0,5	28,2
2	1	28,3
3	1,5	28,5
4	2	28,6
5	2,5	28,3
6	3	28,2
7	3,5	28,4

getIterator(1.5,2);

Figure 6.2: Different iteration type on the same buffer

It has been told that, when sampling is time based, sampling frequency can be selected evaluating the IF-ELSE-EVERY clause. SamplingData class has been implemented to satisfy this need, as well as to represent all the information about sampling timings.

SamplingData class

This class is charged of representing the following information about the sampling:

- The IF-ELSE-EVERY clause which allows to dynamically select sampling frequency;
- The REFRESH clause;
- The behaviour that the system is expected to follow in case of UNSUPPORTED SAMPLE RATE.

The IF-ELSE-EVERY clause is represented using

```
private ArrayList<FrequencyRecord> pRecordSet;
```

where FrequencyRecord is another class representing a single branch of the IF-ELSE-EVERY statement. Each FrequencyRecord contains a frequency value and the condition to be verified to choose the sampling rate. The last entry contained in the ArrayList is relative to the ELSE branch and thus specifies the sampling behaviour that should be used when all the IF conditions are not satisfied. Finally, the other two points of the previous list are represented using the following variables:

```
private UnsupportedSampleRateOption pBehaviour;
private Node pRefreshClause;
```

where the class UnsupportedSampleRateOption is a Java *enum* containing the two currently designed behaviours to follow[2] (i.e.: SLOW DOWN and DO NOT SAMPLE). This class obviously exposes a number of getter/setter methods to access all this information.

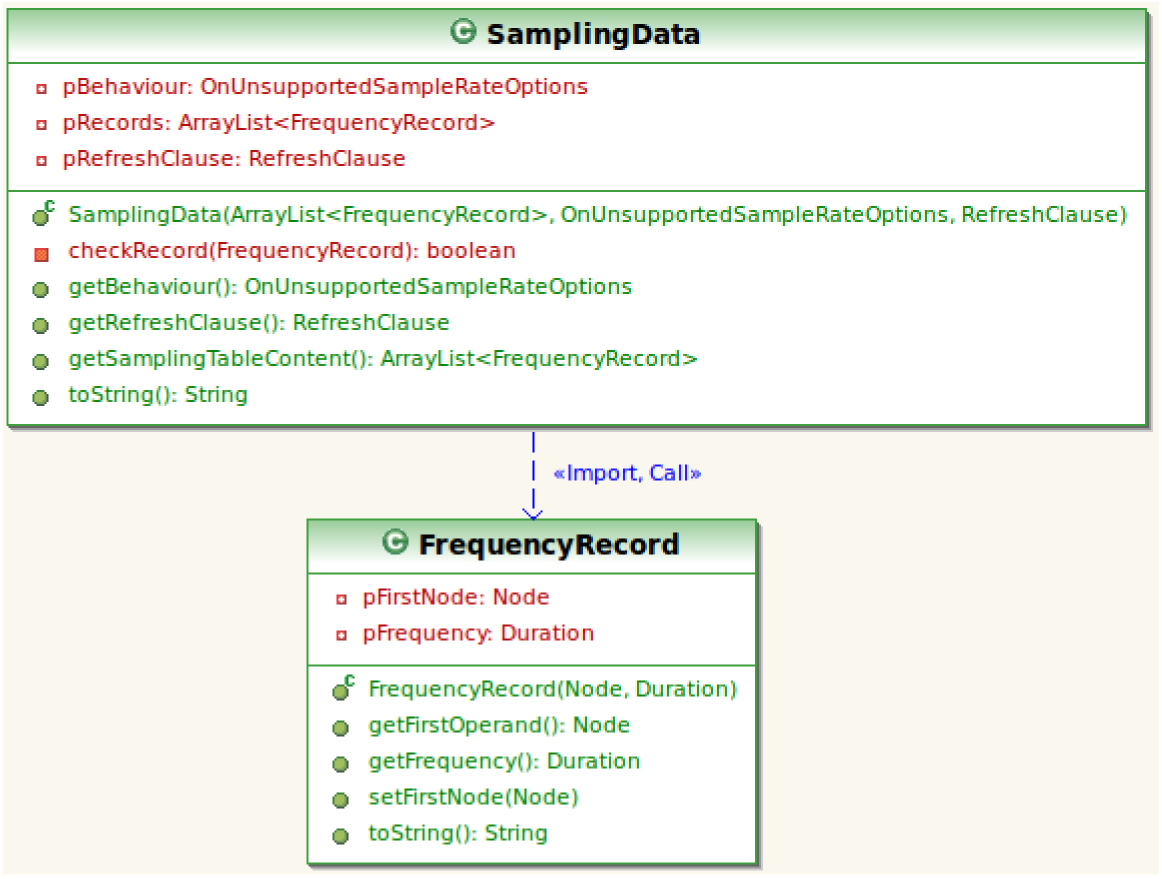

SamplingData
pBehaviour: OnUnsupportedSampleRateOptions
pRecords: ArrayList<FrequencyRecord>
pRefreshClause: RefreshClause
SamplingData(ArrayList<FrequencyRecord>, OnUnsupportedSampleRateOptions, RefreshClause)
checkRecord(FrequencyRecord): boolean
getBehaviour(): OnUnsupportedSampleRateOptions
getRefreshClause(): RefreshClause
getSamplingTableContent(): ArrayList<FrequencyRecord>
toString(): String
«Import, Call»
FrequencyRecord
pFirstNode: Node
pFrequency: Duration
FrequencyRecord(Node, Duration)
getFirstOperand(): Node
getFrequency(): Duration
setFirstNode(Node)
toString(): String

Figure 6.3: UML diagram of the data structures for sampling frequency

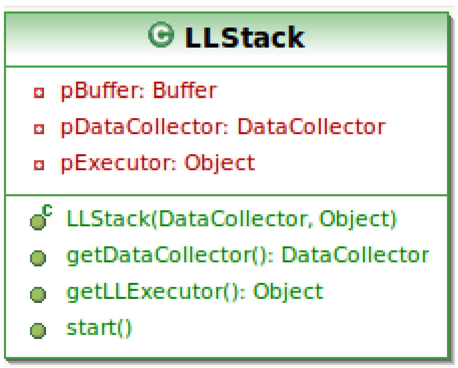

LLStack
pBuffer: Buffer
pDataCollector: DataCollector
pExecutor: Object
LLStack(DataCollector, Object)
getDataCollector(): DataCollector
getLLExecutor(): Object
start()

Figure 6.4: UML diagram of LLStack class

The dissertation now focuses on classes deputed to represent all the information about an LLQ within the Low Level Execution Environment.

LLStack

This class represents the tuple <*DataCollector, Local Buffer, LLQExecutor*> presented in 4.3.1. Its internal state is thus the following:

```
private DataCollector pDataCollector;
private Buffer pLocalBuffer;
private LLExecutor pExecutor[1];
```

The class exposes the usual getter/setter method to retrieve the desired content. Since, as said in Section 4.2, an instance of the DataCollector entity has to be employed for each FPC that is included into query results computation, the number of active tuple (namely LLStack object instances) coincides with the number of FPC returned by the interrogation on the *Local Object Registry*.

[1]Actually this class hasn't been implemented yet, since it's related to the data management part of an LLQ.

QueryInformation class

This class represents the information needed by every query to be successfully computed. It has been chosen to implement this class as *abstract,* grouping here all the aspects that are in common among the three types of query mentioned in Chapter 1. This class, in fact, simply contains a reference to the Statement class [20] whose information are going to be represented. This choice offers the opportunity to extend this class on need, according to the type of query whose information are going to be represented.

In the rest of the dissertation only the class representing information of LLQ are discussed. For completion it can be said that also classes for HLQ and AQ information (respectively HQInformation and AQInformation classes, both inheriting from QueryInformation) have been defined, but their implementation has still to be achieved since this exuled from the scopes of this thesis.

LLQDataCollectionInfo class

This class wraps the "general" information of a single LLQ focusing on information regarding the data collection. It obviously inherits from the QueryInformation class presented above. The reader might ask himself why there's the need of this class, having introduced the SamplingData class. It must then be remembered that the DataCollector entity requires other information to fulfill its objectives further than the information about sampling, including, for example, data regarding the REFRESH clause and data relative to the WHERE clause. More specifically, the class exposes the following information about a LLQ query:

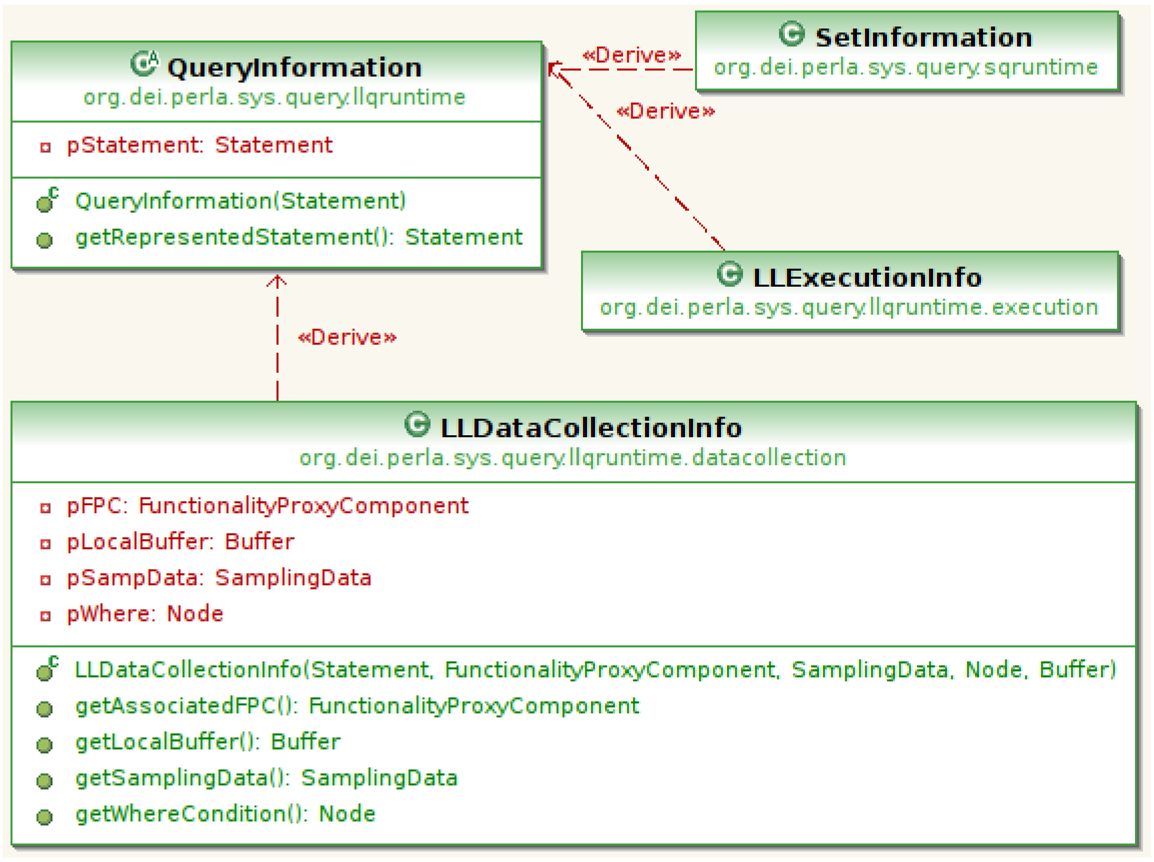
- the expression representing the WHERE clause
- the reference to a SamplingData object (if time based sampling is adopted)
- the reference to the Local buffer where data are meant to be inserted

- the reference to the FPC, where data will come from.

As usual, setter/getter methods have been implemented to allow retrieving all the information mentioned above.

A LLExecutionInfo class has been planned and will be used when the data management part implementation will be completely achieved.

The following Figure represents the relationship between the classes just mentioned above.



QueryInformation
org.dei.perla.sys.query.llqruntime
pStatement: Statement
QueryInformation(Statement)
getRepresentedStatement(): Statement
SetInformation
org.dei.perla.sys.query.sqruntime
«Derive»
«Derive»
LLExecutionInfo
org.dei.perla.sys.query.llqruntime.execution
«Derive»
LLDataCollectionInfo
org.dei.perla.sys.query.llqruntime.datacollection
pFPC: FunctionalityProxyComponent
pLocalBuffer: Buffer
pSampData: SamplingData
pWhere: Node
LLDataCollectionInfo(Statement, FunctionalityProxyComponent, SamplingData, Node, Buffer)
getAssociatedFPC(): FunctionalityProxyComponent
getLocalBuffer(): Buffer
getSamplingData(): SamplingData
getWhereCondition(): Node

Figure 6.5: UML diagrams of classes used to represent information of a query

Environment class

This class can be seen as the container of every query (HLQ, LLQ, AQ) running in the system. Like for QueryInformation, this class has been realized as *abstract*, allowing to represent in one class all the information shared by HLQ, LLQ and AQ as the following subsection explains.

As for QueryInformation class only the classes regarding LLQ are here discussed. Obviously a class inheriting from Environment class has been defined for HLQs and AQs has been defined but its implementation exuled from the purposes of the thesis.

LLQEnvironment class

This class inherits from the previous one and is deputed to wrap every single LLQ running in the system (one instance of the class for every LLQ running). The internal state of this class is the following:

```
private ArrayList<LLStack> pStacks;
```

that's to say all the stacks that are currently employed to compute LLQ results. The class exposes getter/setter methods as can be seen from Figure 6.6.

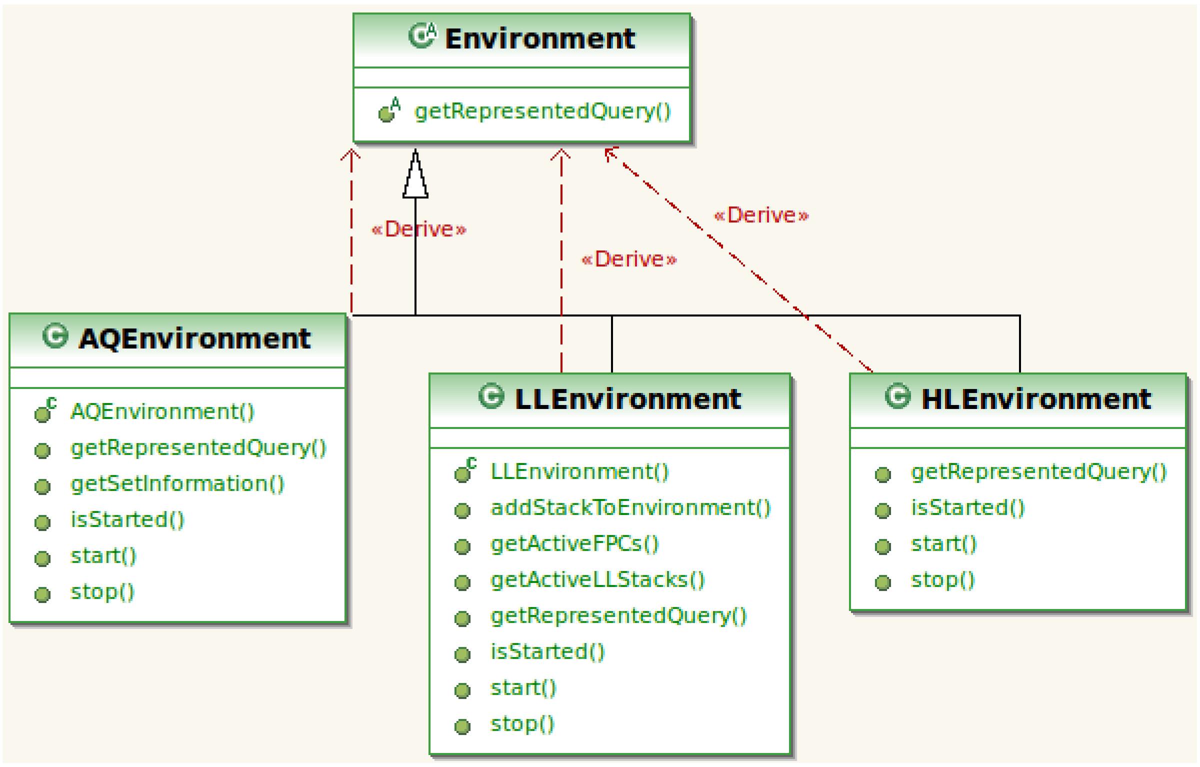


Figure 6.6: LLQEnvironment and LLQInformation UML diagram

ActiveSession class

This class is the most general container of every Environment currently active in the system. It has been defined as static, enabling it to be called without instantiating the object.

For a better comprehension the following figure represents what has been discussed until now, and how the aforementioned classes relates each others.

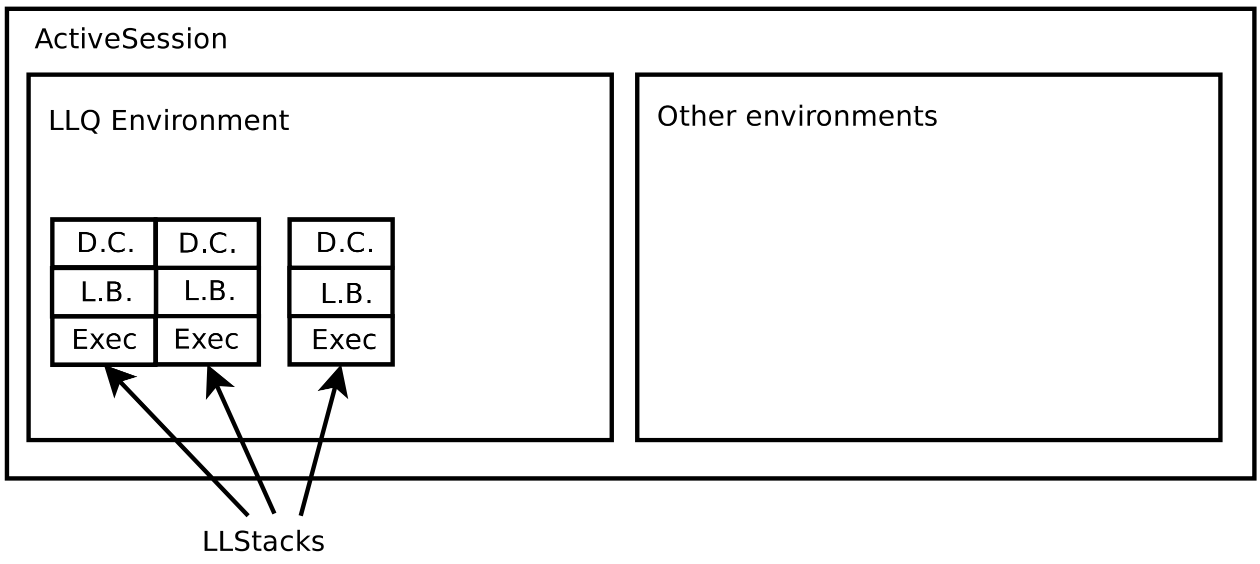


Figure 6.7: ActiveSession, LLQEnvironments and LLStack relationships

6.1.3 Data structure for query analysis

As mentioned in Subsection 3.2.2 Query Analyzer must be invoked to obtain a set of DataCollectors. Query Analyzer is a Java class and is presented hereafter.

QueryAnalyzer class

At this point of dissertation something more must be said about *Query Analyzer*. Actually its activity is not only limited to DataCollector creation. For every PerLa query injected into the system, QueryAnalyzer is charged of analyze it to create all the structures presented before. It essentially performs the following steps:

1. **Query type separation phase**: analyzes the query (analyzing the structure defined in [20]) separating HLQ, LLQ and AQ.

2. **Query information retrieval phase**: For each query type identified a proper

QueryInformation object is built (i.e: LLDataCollectionInfo and LLQExecutionInfo for the LLQs, or HLInformation for HLQs). Notice that for LLQ this step requires the QueryAnalyzer class to query the *Local Object Registry*, in order to obtain a list of the FPCs needed to compute query results.

3. **Environment build phase:** Once the QueryInformation classes are built, the Query Analyzer completes its activity creating an appropriate number of Environment class and adds them to the ActiveSession class.

Obviously the point 2. is the heart of the QueryAnalyzer activity and is hereafter described, focusing on the creation of the LLQDataCollectionInfo class. To achieve the creation of this class and its later insertion in a LLQEnvironment class, the Query Analyzer classes employs a complete set of private methods created ad hoc to analyze queries, each one focusing on a specific aspect. For example getWhereNode() method is used to obtain the WHERE clause, while getSamplingData() is used to obtain the sampling information. Called in the right order this methods allow to build the Query Information classes for each query type identified. It's important to underline that acting this way, the QueryAnalyzer class summarily endorses the role of a *factory*: in fact, considering the whole set of exposed methods, the raw representation of a PerLa query designed in [20] is transformed in a series of structures ready to cooperate to compute query results. This factory fashion is achieved defining the class methods as *static* (enabling them to be called without instantiating a QueryAnalyzer object), and calling the only public method registerQuery() passing as parameter the the Java object representing the root of a query (as defined in [20]). In the following a set of QueryAnalyzer methods is presented, virtually undertaking the three points exposed above, simulating the injection of query. Note that parQuery is always the Java object representing the root of a query. Let's then suppose that a query is injected in PerLa.

1. **Query type separation phase**: this phase starts when the method registerQuery(Query

parQuery) is called. Notice that this is the only public method available in the class, and is used to achieve the factory fashion explained above. Once this method is called, a set of sub-method is called to identify the three query types:

a) createLLEnvironment(Query parQuery);

b) createHLEnvironment(Query parQuery);

c) createSetEnvironment(Query parQuery);

At this point the original problem of analyzing a single query is divided in three independent parts. For each one of the methods exposed, the query information retrieval phase is then activated. This example will focus on what happens after createLLEnvironment(Query parQuery) is called, since this is the case that is relevant in this dissertation.

2. **Query information retrieval phase**: Let's then focus on the method createLLEnvironment(Query parQuery). This method, for each LLQ identified, executes the following steps (always calling proper methods):

   a) It builds the object LLQDataCollectionInfo: this step requires a number of sub-steps:

      i. Querying the *Local Object Registry* in order to receive a list of FPC able to execute the query;

      ii. Building the SamplingData object, which requires to retrieve information about WHERE, REFRESH as exposed in the description of this class.

   b) It builds the object LLQExecutionInfo.

   c) It builds the object LLStack containing a certain number of DataCollector according to the response of the *Local Object Registry*. It must in fact be remembered that a single LLQ can run on multiple FPC, and for each FPC identified a DataCollector must be employed.

3. **Environment build phase:** all the Environment classes built above are then inserted in the ActiveSession class and the computation of the query starts.

As conclusion UML class diagram is reported in Figure 6.8.

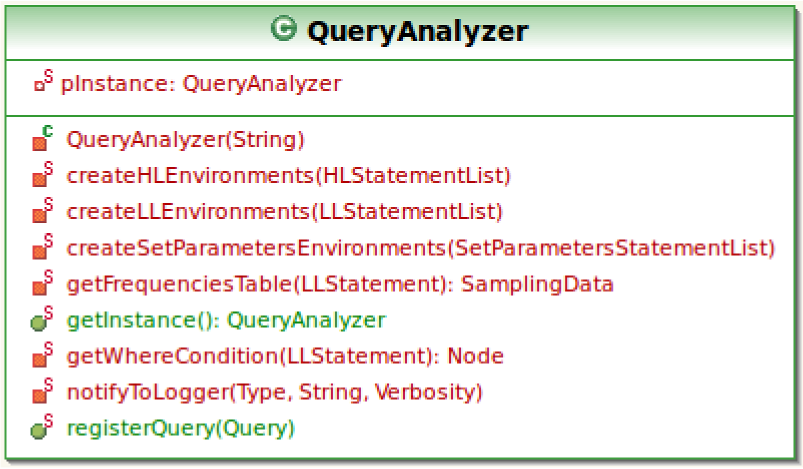

QueryAnalyzer
pInstance: QueryAnalyzer
QueryAnalyzer(String)
createHLEnvironments(HLStatementList)
createLLEnvironments(LLStatementList)
createSetParametersEnvironments(SetParametersStatementList)
getFrequenciesTable(LLStatement): SamplingData
getInstance(): QueryAnalyzer
getWhereCondition(LLStatement): Node
notifyToLogger(Type, String, Verbosity)
registerQuery(Query)

Figure 6.8: UML class diagram of QueryAnalyzer class

ExpressionAnalyzer class

This class is used to analyze expressions within PerLa queries searching for the elements that must be sampled. Till now NodeLogicalObjectAttribute has been introduced as the object which represents an attribute, completely ignoring how such object can be created starting from the user-submitted query. ExpressionAnalyzer can answer this question: since in [17] expressions are implemented as trees (NodeLogicalObjectAttributes or Constants are always leaves, while operators are always Nodes) it uses recursion to navigate through the expression finding NodeLogicalObjectAttributes and returning them to the ExpressionAnalyzer caller.

As shown in Figure 3.7 this class exposes only one public method:

```
public ArrayList<NodeLogicalObjectAttribute> getNodeArrayList (Node parNode);
```

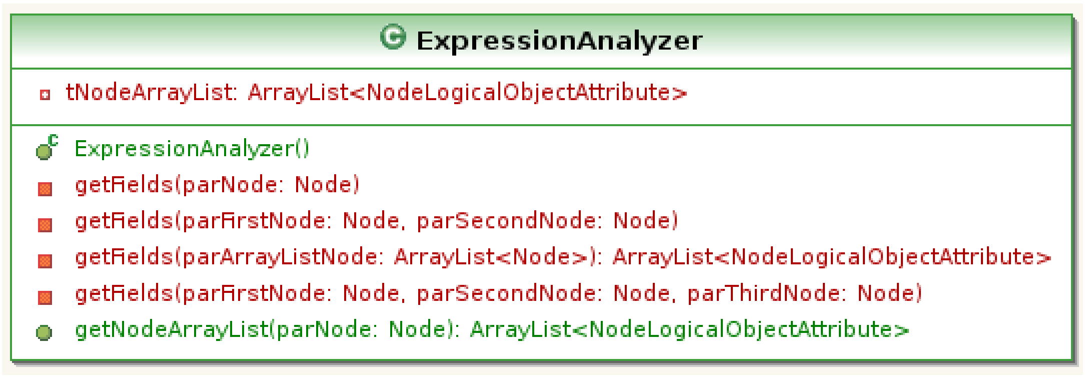

ExpressionAnalyzer
tNodeArrayList: ArrayList<NodeLogicalObjectAttribute>
ExpressionAnalyzer()
getFields(parNode: Node)
getFields(parFirstNode: Node, parSecondNode: Node)
getFields(parArrayListNode: ArrayList<Node>): ArrayList<NodeLogicalObjectAttribute>
getFields(parFirstNode: Node, parSecondNode: Node, parThirdNode: Node)
getNodeArrayList(parNode: Node): ArrayList<NodeLogicalObjectAttribute>

Figure 6.9: ExpressionAnalyzer UML diagram

where parNode is the root of the expression Java representation. According to the number of children of the root, a private method (getFields(Node parNode)) is called on each child. This method checks if a Node is an instance of NodeLogicalObjectAttribute class and, if so, appends it to a Java ArrayList (checking and avoiding multiple insertion of the same NodeLogicalObjectAttribute). Instead, if the child is an internal node, getFields(Node parNode) calls recursively a proper method, depending on the number of children of the internal node:

```
public void getFields (Node parFirstNode, Node parSecondNode);

public void getFields (Node parFirstNode, Node parSecondNode, Node parThirdNode);

public void getFields (ArrayList<Node> parArrayListNode);
```

Such methods differ each others only for the number of children they deal with, but they work in the same way: for each child getFields(Node parNode) is called again on all its children, generating the recursion mentioned above.

ExpressionEvaluator class

This is the static class which physically computes expression evaluation. The static method

```
public static boolean evaluateExpression(Record parRecord, Node parExpression);
```

uses ExpressionAnalyzer to retrieve the set of NodeLogicalObjectAttribute from parExpression and substitutes any recurrence of the obtained set with their counterparts contained in parRecord, which is a record of sampled values. Such substitution is achieved filling the pValue field of NodeLogicalObjectAttribute (as described in Subsection 6.1): at this point attributes have been substituted and expression computation can be executed calling the getResult() method of Node class, as implemented in [17]. The (boolean) result of the evaluation is then returned to the caller.

6.2 Data collection implementation

6.2.1 Communication structure implementation

Pipe class

Pipe class plays the role of a **synchronized** channel between two elements. Referring to Figure 6.9 let's explain how synchronization is achieved, supposing that component A has to send a message for component B.

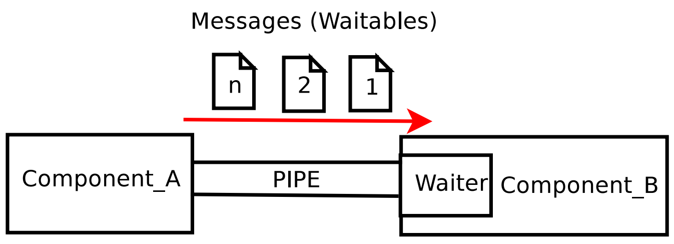


Figure 6.10: Pipes

Since component B needs to wait for messages at the end of the pipe, a Waiter object has been defined, whose main role is to wait for the next element "traveling" on the pipe: such messages are called Waitable objects. When Waiter notices that a Waitable object reaches the end of the pipe, it signals that a new message has arrived to its final destination: at this point the message can be removed from the pipe and sent to upper levels for further modifications. At this time the Waiter begins again waiting for next message on the pipe. It must be noticed that more than one pipe can be linked with the same Waiter: in fact, one component can be interested in receiving messages from more than one component at the same time. Synchronization is then a key element in Pipe, and it is insured by enqueue() and dequeue() methods implemented within Pipe class: Waiter, in fact, must manage a certain number of messages on multiple pipes, always avoiding data loss. Pipes, then, allow two communicating PerLa elements to be decoupled: calling standard ad-hoc methods to transfer messages introduces the risk of "freezing" the system if a message is still not ready when the method is called, thus deteriorating global performances. Waiter has been designed exactly to allow avoiding such a risk: instead of physically retrieving a message, message itself is simply waited.

The following figure shows the flow of data after introducing the concept of *pipes*.

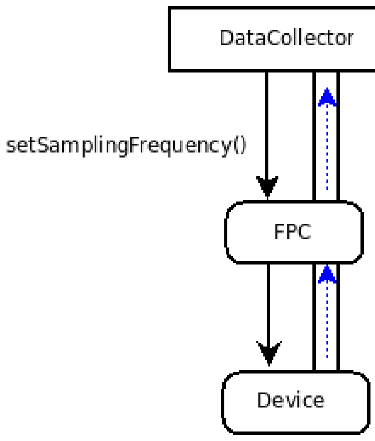


Figure 6.11: Data flow using pipes

It's now possible to discuss the class implemented to achieve the data collection, the DataCollector class, whose design was proposed in 4.3.1

6.2.2 DataCollector class

DataCollector lifecycle as Java object begins with its constructor, which simply needs to receive the LLDataCollectionInfo object as parameter. DataCollector class has been written extending Java Thread Class, overriding run() method to implement data collection activity. The first step DataCollector takes is waiting (instantiating a Waiter) a first record. Waiting a record requires DataCollector to open a pipe with FPC, calling the method getNewOutputPipe() method on the FPC reference contained in the object LLDataCollectionInfo received by the constructor. It's important to highlight that the use of pipes allows DataCollector to manage time-based and event-based samplings in the same way: Waiter, in fact, completely ignores which is the cause that generated the record it receives. This can be easily understood by viewing "time based" sampling as a particular type of "event based" sampling, with the only difference that in a "time based" sampling the arrival instants of records are predictable (according to the adopted sampling frequency), while they're completely aleatory in the "event based" mode. This particular characteristic allows DataCollector implementation to manage the two sampling types using the same portion of code. DataCollector is anyhow aware of sampling types difference and, when in time based mode, it manages the setting of the sampling frequency as well as the REFRESH clause. This task is achieved by a dedicate private (thus internal) method called evaluateFrequency(). This method uses the first record received from the pipe to set the sampling frequency on the device. The reader might be surprised that it's necessary to wait for a first record to evaluate sampling frequency. Instead, it's perfectly reasonable since evaluating the sampling frequency requires the evaluation of the IF branch in the IF-EVERY-ELSE clauses: without a record of sampled data this couldn't be possible. After that the sampling has been chosen, the evaluateFrequency() method calls setSamplingFrequency() method on its relative FPC. Next sampling frequency updates are instead delegated to the information contained into the REFRESH clause (if present). This task is accomplished with the definition of an inner class called RefreshHandler. This class extends the Java Thread class, and is able to read the REFRESH clause frequency contained into the LLDataCollectionInfo object. According to this frequency, this class class calls forceSampling() method exposed by the FPC, in order to obtain a set of sampled data. With these records DataCollector is able to calculate the new sampling frequency and to set it calling the aforementioned setSamplingFrequency() method. According to Figure 4.5, next step to be performed is then the evaluation of the WHERE clause using ExpressionEvaluator class: if it is satisfied, the record is inserted in the local buffer (parBuffer) otherwise it's discarded and terminate condition finally discriminates if the cycle has to be repeated again or data collection activity has terminated. A do/while block allows the whole cycle to be repeated as many time as needed, until data collection activity ends.

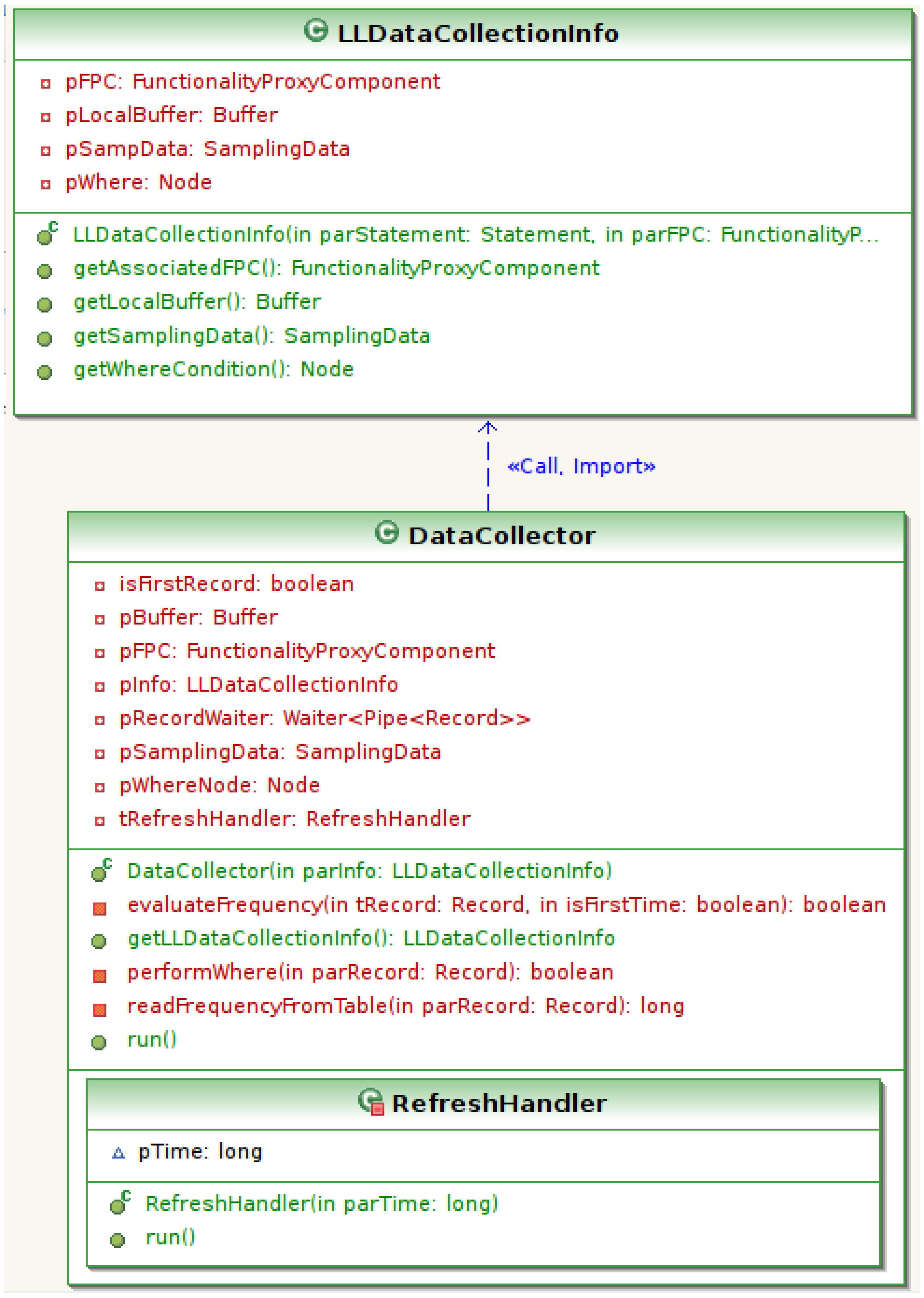

LLDataCollectionInfo
pFPC: FunctionalityProxyComponent
pLocalBuffer: Buffer
pSampData: SamplingData
pWhere: Node
LLDataCollectionInfo(in parStatement: Statement, in parFPC: FunctionalityP...
getAssociatedFPC(): FunctionalityProxyComponent
getLocalBuffer(): Buffer
getSamplingData(): SamplingData
getWhereCondition(): Node
«Call, Import»
DataCollector
isFirstRecord: boolean
pBuffer: Buffer
pFPC: FunctionalityProxyComponent
pInfo: LLDataCollectionInfo
pRecordWaiter: Waiter<Pipe<Record>>
pSamplingData: SamplingData
pWhereNode: Node
tRefreshHandler: RefreshHandler
DataCollector(in parInfo: LLDataCollectionInfo)
evaluateFrequency(in tRecord: Record, in isFirstTime: boolean): boolean
getLLDataCollectionInfo(): LLDataCollectionInfo
performWhere(in parRecord: Record): boolean
readFrequencyFromTable(in parRecord: Record): long
run()
RefreshHandler
pTime: long
RefreshHandler(in parTime: long)
run()

Figure 6.12: DataCollector UML class diagram

7 FPC interface implementation

This chapter will focus on the implementation aspects regarding the FPC interface. Since the interface will be used to link together the LLQ Execution with PerLa lower levels it has been written in Java and consists in two Java classes:

- FunctionalityProxyComponent
- FPCDataStructures

Each class is going to be discussed in the following.

7.1 FPCDataStructure

This class contains a series of fundamental information needed by the FPC. The most important is surely the couple

```
public final HashMap<Constant, String parName> pAttributes;

public abstract Constant getAttributeByName(String parName);
```

This couple is used to identify what attributes are abstracted by a the FPC component and are used by the Local Object Registry to catalog the FPC, in order to later decide if an FPC must participate into a query results computation.

7.2 FunctionalityProxyComponent class

This is the class implementing the designed interface. This is why it has been created as abstract as well as the methods inserted. This class must in fact must be extended and widened dynamically by the FPC factory component according the provided XML file as explained in previous chapters. Let's start the description of the class with the internal data structures:

```
private final FPCDataStructure pFPCfinalStructure;
```

The FPCDataStructure is the object presented in the previous section.

```
private final Pipe<AdapterFpcMessage> inputPipe;

private final Waiter<Pipe<AdapterFpcMessage>> inputPipeWaiter;
```

are respectively the Pipe and Waiter objects presented in the Section 6.2.1, and are used as channels to receive data from the sensor (or sensors) that the FPC is abstracting. The Waitable object travelling through these Pipe objects is another Java object called AdapterFpcMessage. This class is part of the complex framework that allows FPC to communicate with the abstracted device and more details can be found in [7, 8]. For this dissertation this object can be considered as a Java object encapsulating the value(s) sampled by the abstracted sensor.

```
private final ArrayList<Pipe<Record>> outputPipeArrayList;
```

This ArrayList contains the collection of Pipe objects that are used to link FPC to the DataCollectors. The collection is necessary since it must be remembered that an FPC could be involved into the execution of more than one LLQ (and thus, more than one DataCollector). When a DataCollector requires a new pipe it calls the getNewOutputPipe()

method which has been included in this class.

Before the FPC can start operating, the LLQ that is going to be served must be registered on the FPC using the method registerQuery(). In the same way when an LLQ has terminated this method counterpart must be invoked calling unregisterQuery().

Once the LLQ has been registered values can be read from the device using a dedicated method called readMessageFromAdapter(). This method has obviously been left abstract since every device generates a record according to a set of rules typically decided by the producer. A typical example is the endianess of the bytes representing a sampled value: some producers use Little Endian, while others prefer to use Big Endian. Another example lies in the number of bits that are adopted to represent an integer value (8,16 or 32 usually): the number varies from producer to producer. Since it's the FPC component that complies with devices rules and not viceversa, the readMessageFromAdapter() method must be extended from time to time by the FPC factory according to device abstracted.

The last important method is forceSampling(). This method is the implementation of the desired ability of the FPC to withdraw a sampled value from the pervasive system to reevaluate sampling frequencies. This is the only case, and the only reason in PerLa where it is allowed to obtain a sampled value with the direct call of a method instead of using the excellent decoupling method provided by pipes. Once a new frequency has been selected it can be set using the method setSamplingFrequency(). The last two presented methods are again left abstracted: again every device has its proper rule and these methods must be expanded and filled by the FPC factory component.

The class finally includes start(), stop() and isStarted() methods whose use is evident. In the following figure the complete UML diagram is reported.

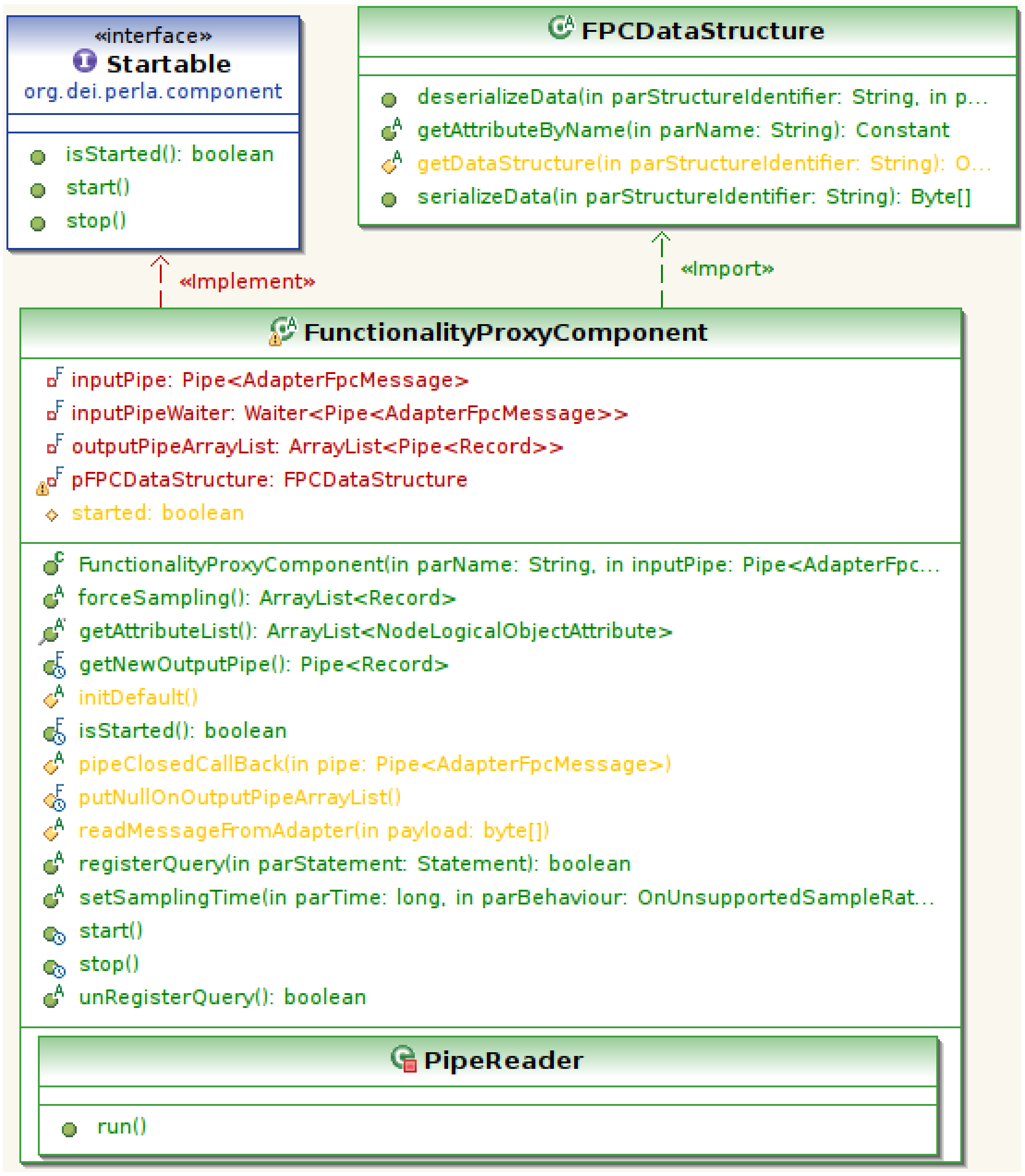

«interface»
Startable
org.dei.perla.component
isStarted(): boolean
start()
stop()
FPCDataStructure
deserializeData(in parStructureIdentifier: String, in p...
getAttributeByName(in parName: String): Constant
getDataStructure(in parStructureIdentifier: String): O...
serializeData(in parStructureIdentifier: String): Byte[]
«Implement»
«Import»
FunctionalityProxyComponent
inputPipe: Pipe<AdapterFpcMessage>
inputPipeWaiter: Waiter<Pipe<AdapterFpcMessage>>
outputPipeArrayList: ArrayList<Pipe<Record>>
pFPCDataStructure: FPCDataStructure
started: boolean
FunctionalityProxyComponent(in parName: String, in inputPipe: Pipe<AdapterFpc...
forceSampling(): ArrayList<Record>
getAttributeList(): ArrayList<NodeLogicalObjectAttribute>
getNewOutputPipe(): Pipe<Record>
initDefault()
isStarted(): boolean
pipeClosedCallBack(in pipe: Pipe<AdapterFpcMessage>)
putNullOnOutputPipeArrayList()
readMessageFromAdapter(in payload: byte[])
registerQuery(in parStatement: Statement): boolean
setSamplingTime(in parTime: long, in parBehaviour: OnUnsupportedSampleRat...
start()
stop()
unRegisterQuery(): boolean
PipeReader
run()

Figure 7.1: FPC UML diagrams

8 Conclusion and open points

The objectives of design proposed at the beginning of the dissertation have been fully satisfied with the introduction of the *DataCollector* and *LLQExecutor* entities and the definition of a precise interface with lower levels, followed by the implementation of the proposed design.

Some point still remains open:

1. The *LLQExecutor* entity has been designed but still needs to be implemented. Future developments will focus on this aspect.

2. Following to point 1, QueryAnalyzer class still requires the implementation of a certain number of methods to build LLExecutionInfo class as well as all the other structure that will be needed. (e.g. to retrieve the set of fields needed to compute SELECT, HAVING and GROUP BY clauses). These functionalities were, in fact, out of the thesis scopes.

3. ExpressionEvaluator class is still limited in evaluating a condition using one single record. A complete implementation must be achieved when data management section will completed, since it deals with aggregates (e.g. AVG, MIN, GROUP BY, etc) whose computation requires more than one single record to be performed.

4. *Logical Object Registry* still requires to be completely designed and implemented, as briefly mentioned in this dissertation.

Future developments should provide answers to the previous questions. Finally the whole stack should be tested in a real scenario: PerLa is in fact currently scheduled to monitor rockfalls in Lecco (MI) within Prometeo project [3].

Appendix A: PerLa console

The need of remote control

During the implementation phase of this thesis it came clear the need to be able to remotely control an instance of PerLa. The need arises from a series of considerations, among the others:

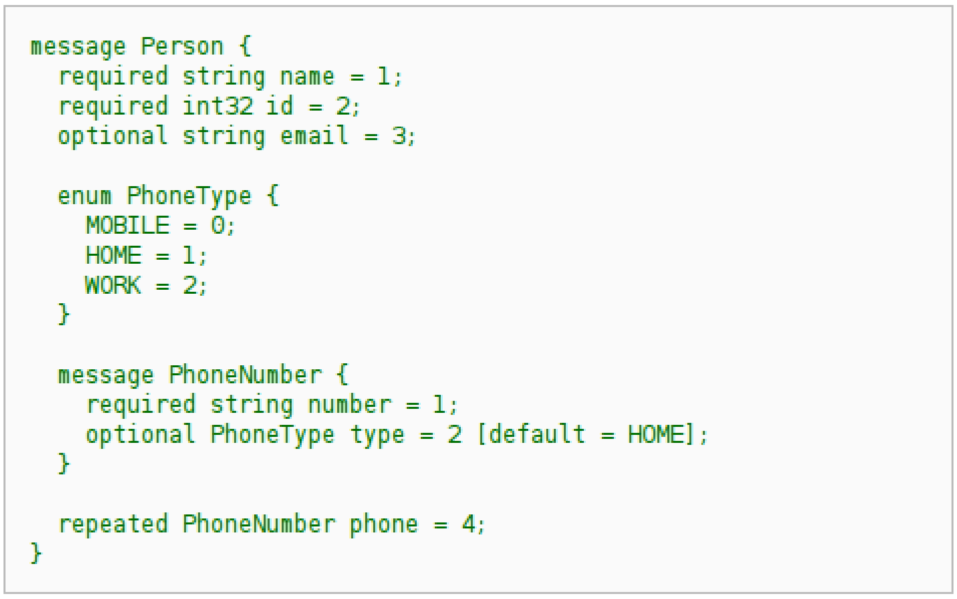
1. PerLa is currently under continuous development: implementing a new component, or modifying an existing one always requires a test phase which includes to test component interactions with the others. In PerLa almost every test is conducted by simulating the injection of a query prototype in order to see if the system reacts as it is designed to. Again this process requires to manually create a Java executable test class deputed at creating all the objects necessary (parser, Logical Object Registry, Query Analyzer, Environments and so on) as well as running the test itself. The whole process needed to be sped up and easily performable.

2. In most of PerLa real implementation the final query results (i.e. the results computed by the HLQs) are not directly exposed to the user but are "redirect" to other applications for further modifications, for example MatLab®.

3. Some PerLa instances operates in physical inaccessible places (or hardly accessible) by human beings. An example is reported in [3], where the sensors are placed on a side of mount S.Martino in Lecco (Italy) well known for rockfalls phenomena. In

case of necessary modifications (upgrades, fails recovers) the need to control remotely or to upload/download some data would make the operation free of risks and easily performable.

The component deputed to this important task is called PerLa *console,* since it allows to execute a certain number of remote commands. The console is, however, no more than a normal client/server application: the server runs on the same machine that is currently hosting a PerLa instance, while the client side of the application uses TCP/IP to connect to the server in order to send commands and receive responses. The "critical" design phase was then about the format to be adopted for the messages that client and server exchange each others. Google Inc. has been developing, during recent years, an infrastructure called *Protocol Buffers* [4] which made the first console implementation to be ready in few days. Protocol Buffers (shortened ProtoBuf) are a flexible, efficient and automated mechanism for serializing structured data, and is currently used by Google itself for almost its internal RPC protocols. ProtoBuf allow to define how messages are to be structured using a very simple, but powerful grammar as exposed in Figure:



```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Figure 8.1: ProtoBuf grammar, an example taken from [4]

The message structure written with this grammar is usually saved in text files called "proto files" (the names derives from the file extension, `.proto`)

Google than provides a ProtoBuf parser for these files that is able to create the Java, C++ and Python classes that implement the desired structure. The big advantage is evident: instead of spending time in implementing classes, data structures, access methods for the desired message, ProtoBuf allows to simply write the desired structure: the classes created after the parsing phase already contains all the necessary to operate (constructors, methods, and so on). The advantage is maybe more evident when the messages exchanged need to be modified: ProtoBuf allow to update the desired message without breaking deployed programs that are compiled using the "old" method.

PerLa Console using Google ProtoBuf

The messages exchanged between server and client are two:

Request it "travels" from client to server and typically consists in a command to be executed

Response is the response to a received command

In both of these messages, the internal structure is composed as exposed in the following figure:

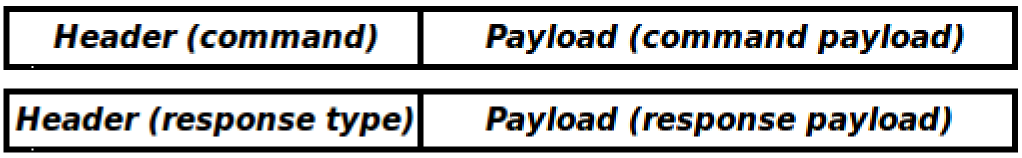


Figure 8.2: PerLa console messages structure

The *header* is used to transmit from client to server the desired command to be executed, and from server to client the response type (i.e: the (un)successful execution of the command). Since it was necessary to obtain an implementation of the console that could help PerLa development (as exposed in the first point of the previous section) the following commands were the first to be included and chosen:

TEST_QUERY it's supposed to be used to test a query without injecting it into the system, in order to check its syntax

INJECT_QUERY to remotely inject a query

SHOW_REGISTRY to explore the content of the Logical Object Registry

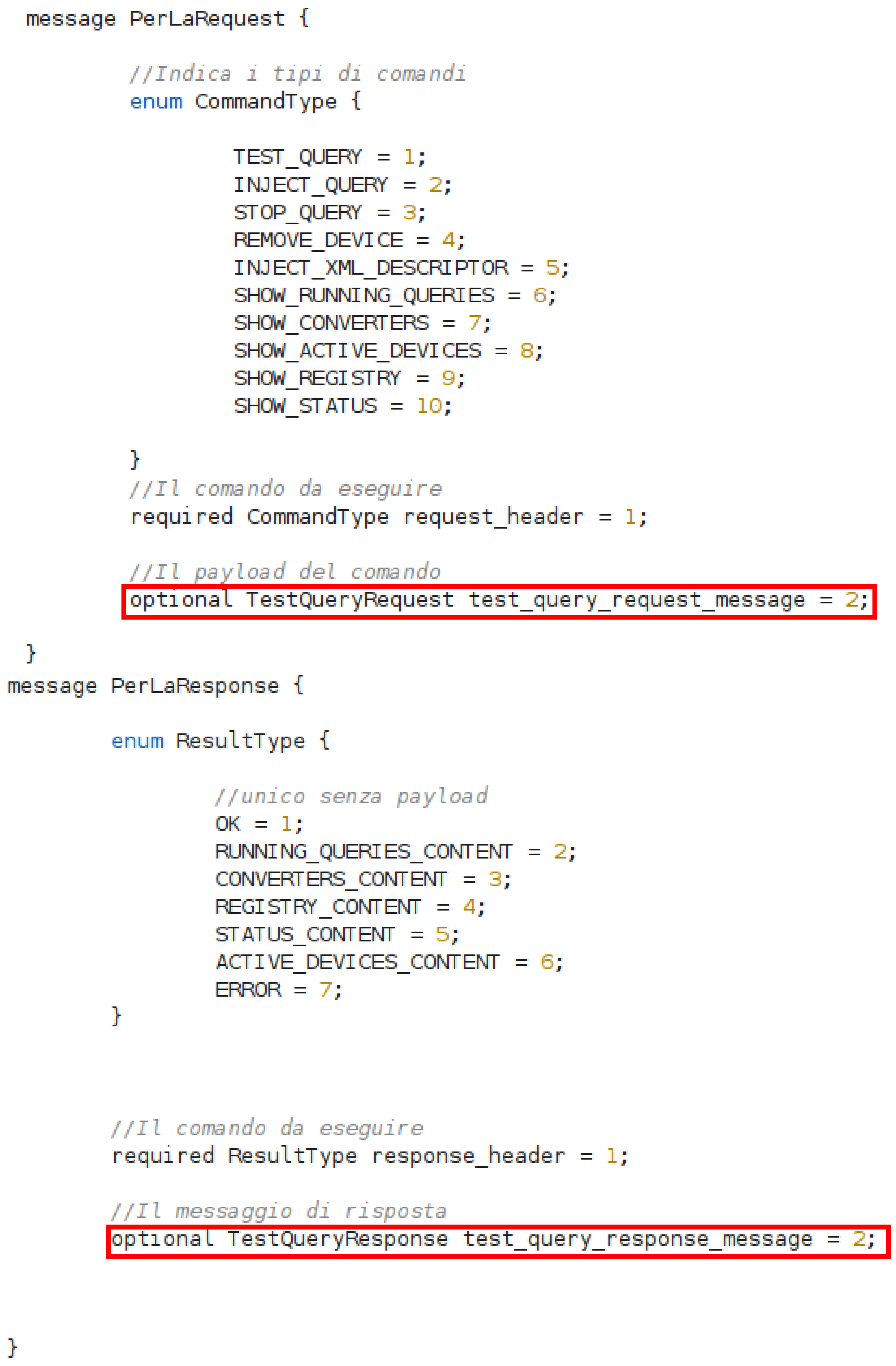
SHOW_QUERY to retrieve all the information about a running query

STOP_QUERY to stop a running query

SHOW_DEVICES to retrieve information about the added devices

INJECT_DESCRIPTOR to manually provide PerLa with the XML file in order to add a device that is not able to provide autonomously the XML file.

Every command may request the presence of a proper *payload:* this is why a second section within message structure has been reserved. For example the command **TEST_QUERY** requires the textual query that must be checked. In the same way the response payload will contain the confirmation of the correctness of the query, or the syntactic error discovered. The message structure (header and payload considered together) of both requests and responses were defined using the following proto file (perlamessage.proto). Their structures using ProtoBuf are exposed in the following figures:

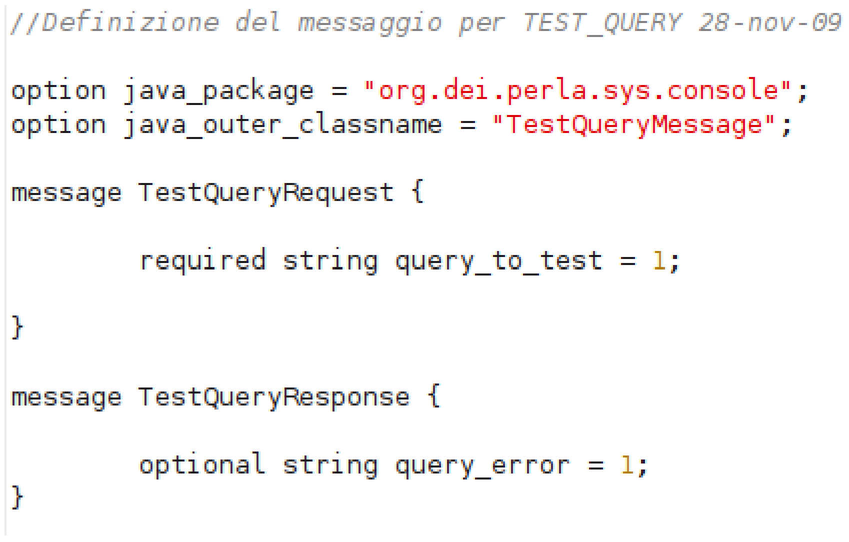


```
message PerLaRequest {

        //Indica i tipi di comandi
        enum CommandType {

                TEST_QUERY = 1;
                INJECT_QUERY = 2;
                STOP_QUERY = 3;
                REMOVE_DEVICE = 4;
                INJECT_XML_DESCRIPTOR = 5;
                SHOW_RUNNING_QUERIES = 6;
                SHOW_CONVERTERS = 7;
                SHOW_ACTIVE_DEVICES = 8;
                SHOW_REGISTRY = 9;
                SHOW_STATUS = 10;

        }
        //Il comando da eseguire
        required CommandType request_header = 1;

        //Il payload del comando
        optional TestQueryRequest test_query_request_message = 2;

}
message PerLaResponse {

        enum ResultType {

                //unico senza payload
                OK = 1;
                RUNNING_QUERIES_CONTENT = 2;
                CONVERTERS_CONTENT = 3;
                REGISTRY_CONTENT = 4;
                STATUS_CONTENT = 5;
                ACTIVE_DEVICES_CONTENT = 6;
                ERROR = 7;
        }


        //Il comando da eseguire
        required ResultType response_header = 1;

        //Il messaggio di risposta
        optional TestQueryResponse test_query_response_message = 2;


}
```

Figure 8.3: Request and response messages definition with ProtoBuf

ProtoBuf grammar enables the user to choose if a message has to contain mandatory or optional fields (required and optional keywords). In a request the command is obviously mandatory while, as said before, the payload is optional. The classes dynamically created by ProtoBuf automatically raise a Java Exception if a mandatory field is found empty: this is another advantage inherent to the use of ProtoBuf instead of manually coding Java classes.

The fields highlighted in red are the definitions of the payloads and require a further discussion. The payload of every request/response must have a precise structure, and is defined using ProtoBuf again. In this case the payload of the request containing command **TEST_QUERY** is defined in a message called TestQueryRequest, while the payload response is a message called TestQueryResponse. Both messages structures are reported here below, since they're written in the same proto file:



```
//Definizione del messaggio per TEST_QUERY 28-nov-09

option java_package = "org.dei.perla.sys.console";
option java_outer_classname = "TestQueryMessage";

message TestQueryRequest {

        required string query_to_test = 1;

}

message TestQueryResponse {

        optional string query_error = 1;
}
```

Figure 8.4: TestQueryResponse and TestQueryRequest payload definition

It's now clear how all the messages are defined. A first proto file describes the structure of a generic request/response and contains mandatory the command to execute, or the response to a command. If the command/response requires a payload it must be defined into a separate proto file that will contain the definition of the request/response payloads. In other words there is only one proto file to describe request and responses while there's a single file, describing eventually the payloads, for each command presented at the beginning of this section.

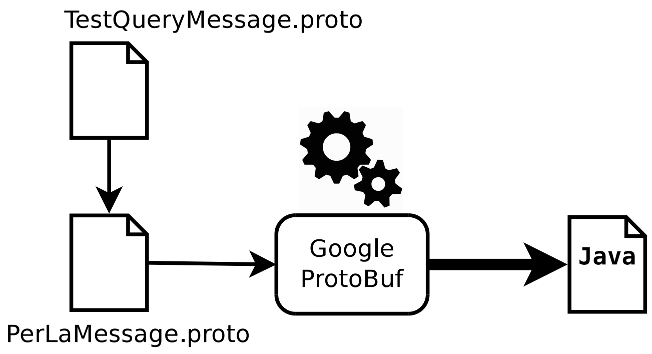


Figure 8.5: PerLa console functioning schema

Actual and future works

Currently only the proto files for the request/response messages and for the **TEST_QUERY** command has been written and tested. The console is, in fact, still at an primitive level, as well as the list of executable commands. The commands themselves are also in a very simple form. Future works should consolidate the current design, defining a precise grammar for the console also considering all the possible application scenarios.

Appendix B: Source code

In the following the source code of the most important implemented class is proposed.

DataCollector class

```
package org.dei.perla.sys.query.llqruntime.datacollection;

import org.dei.perla.parser.clauses.*;
import org.dei.perla.parser.expressions.Node;
import org.dei.perla.sys.device.fpc.FunctionalityProxyComponent;
import org.dei.perla.sys.query.llqruntime.datastructures.*;
import org.dei.perla.utils.pipe.*;
import org.dei.perla.utils.waiter.*;

/**
 * Questa classe rappresenta l'oggetto incaricato del recupero dei dati campionati
     da parte
 * dell'<tt>FPC</tt>.
 */
public class DataCollector extends Thread{


  /**
   * Tutte le informazioni riguardo l'attivita' di data collecting.
   */
  private LLDataCollectionInfo pInfo;
  private FunctionalityProxyComponent pFPC;
  private Buffer pBuffer;
  private SamplingData pSamplingData;
  private Waiter<Pipe<Record>> pRecordWaiter;
  private Node pWhereNode;


  /**
   * Flag per marcare la situazione "primo record ricevuto"
   */
  private boolean isFirstRecord = true;

  /**
   * Flag per marcare che e' gia' attivo un gestore di REFRESH
   */
```

```
private RefreshHandler tRefreshHandler = null;


/**
 * Oggetto che si occupa di gestire la clausola REFRESH
 */
private class RefreshHandler implements Runnable {

  long pTime;

  public RefreshHandler(long parTime) {
    this.pTime = parTime;
  }

  public void run() {
    try {
      sleep(pTime);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    pFPC.setSamplingTime(readFrequencyFromTable(pFPC.forceSampling().get(0)),
        pSamplingData.getBehaviour());
  }
}

/**
 * Costruisce un nuovo oggetto del tipo DataCollector
 * @param parInfo le informazioni di cui ha bisogno il DataCollector
 */
public DataCollector(LLDataCollectionInfo parInfo) {

  this.pInfo = parInfo;
  this.pBuffer = parInfo.getLocalBuffer();
  this.pFPC = parInfo.getAssociatedFPC();
  this.pSamplingData = parInfo.getSamplingData();
  this.pWhereNode = parInfo.getWhereCondition();
  pRecordWaiter = new Waiter<Pipe<Record>>();
  pRecordWaiter.start();
  pRecordWaiter.addWaitable(pFPC.getNewOutputPipe().getWaitable());
  //pRecordWaiter.addWaitable(((FPCmockup) pFPC).getPipe((LLStatement) parInfo.
      getRepresentedStatement()).getWaitable());
}

/**
 * @return Le informazioni di basso livello associate a questo DataCollector
 */
public LLDataCollectionInfo getLLDataCollectionInfo() {
  return this.pInfo;
}

/**
 * Mette in esecuzione il DataCollector
 */
public void run() {

  Record tRecord;
```

```
    System.out.println("*** DATA COLLECTOR starts ***");
    Waitable<Pipe<Record>> tTmpWaitable;
    Pipe<Record> tTmpPipe;

    //....ciclo principale di raccolta dati

    do {
      tTmpWaitable = pRecordWaiter.waitNext();
      tTmpPipe = tTmpWaitable.getParentComponent();
      tRecord = tTmpPipe.dequeue();

      if(tRecord == null) {
        System.out.println("PIPE CHIUSA");
        tTmpPipe.stop();
        pRecordWaiter.stop();
        tRecord = null;
        break;
      }

      System.out.println("Record letto da "+pFPC.getName());

      if (performWhere(tRecord)) {
        //...i record che si "salvano" dopo la performWhere() vengono inseriti nell
            'LocBuf
        pBuffer.addRecord(tRecord);
        System.out.println("Record aggiunto dal DC collegato a "+pFPC.getName());
        System.out.println("["+pBuffer.getBuffer().size()+"]"+"<--- Il record ora
            nel buffer contiene il valore: \n" + tRecord);
      }

      else {
        System.out.println("*** Record SCARTATO dalla WHERE***");
      }


      //...valuto se Ãš il caso di cambiare la frequenza di campionamento....
      evaluateFrequency(tRecord, isFirstRecord);
      isFirstRecord = false;

      //TODO inserire qui la gestione della terminate (basta un "break")


    } while (true);

    System.out.println("*** DATA COLLECTOR ha terminato ***");
    //TODO bisogna inserire qui tutta le gestione della terminazione della query
  }

  /**
   * Metodo helper: data una tabella e un record aggiorna la frequenza di
       campionamento sull'FPC
   * @param parTable la tabella
   * @param parRecord il record
   * @return TRUE se ok, FALSE altrimenti
   */
  private long readFrequencyFromTable(Record parRecord) {
    long foundFrequency = 0;

    for (FrequencyRecord tFreqRec : pSamplingData.getSamplingTableContent()) {

      if (ExpressionEvaluator.evaluate(parRecord,tFreqRec.getFirstOperand())) {
        foundFrequency = (long) tFreqRec.getFrequency().getValue();
        break;
        //bValue = ((FPCmockup) pFPC).setMockupSamplingTime((long)tFreqRec.
            getFrequency().getValue()*1000,null,(LLStatement) pInfo.
            getRepresentedStatement());
        //bValue = (pFPC.setSamplingTime((long)tFreqRec.getFrequency().getValue()
            *1000, parTable.getBehaviour(),pInfo.getRepresentedStatement()));
        //if (bValue == true) break;
      }

    }
    return foundFrequency;
  }
  /**
   * Metodo helper per la valutazione della clausola Where
   * @param parRecord il record su cui controllare la clausola WHERE
   * @return TRUE se il record Ãš accettabile, FALSE se va' scartato
   */
  private boolean performWhere(Record parRecord) {

    return ExpressionEvaluator.evaluate(parRecord,pWhereNode);
  }

  /**
   * Metodo interno che si occupa della parte riguardante le tempistiche e la
       casistiche di sampling
   * @param tRecord
   * @param isFirstTime indica se e' la prima volta che il DC setta la frequenza.
       Infatti il primo record viene usato
   * per settare la frequenza. Al giro successivo e' compito della REFRESH
       occuparsene.
   * @return TRUE se tutto ok, FALSE otherwise.
   */
  private boolean evaluateFrequency(Record tRecord, boolean isFirstTime) {

    /*E' il primo record che e' stato ricevuto?*/

    if (isFirstTime) {
      //...allora setto la prima frequenza di campionamento..
      this.pFPC.setSamplingTime(readFrequencyFromTable(tRecord),pSamplingData.
          getBehaviour());
    }
    else {
      //...a questo punto almeno un settaggio della FC (frequenza di campionamento)
          Ãš stato fatto.
      //..devo vedere se Ãš il caso di fare REFRESH e che tipo di REFRESH:

      if (pSamplingData.getRefreshClause() instanceof RefreshNever) {
        //..l'utente ha imposto nessun REFRESH... la prima frequenza impostata
            rimane per sempre.
      }
      else if (pSamplingData.getRefreshClause() instanceof RefreshEventBased) {

        //Qui e' tutto un open point. Dal mio punto di vista io farei cosi':
        //1. aggiungerei interfaccia all'FPC per un'altra pipe di Eventi
        //2. aspetto gli eventi che salgono e me li segno
        //3. se tra quelli che ho segnato c'Ãš quello che mi interessa...aggiorno
            ...

      }
      else if (pSamplingData.getRefreshClause() instanceof RefreshTimeBased) {

        if (tRefreshHandler == null) {
          tRefreshHandler = new RefreshHandler((long) ((RefreshTimeBased)
              pSamplingData.getRefreshClause()).getInterval().getValue());
          tRefreshHandler.run();
        }

      }
    }
    return true;
  }
}
```

FunctionalityProxyComponent class

```
package org.dei.perla.sys.device.fpc;

import java.util.*;
import org.dei.perla.component.LoggableComponent;
import org.dei.perla.component.Startable;
import org.dei.perla.parser.clauses.OnUnsupportedSampleRateOptions;
import org.dei.perla.parser.expressions.NodeLogicalObjectAttribute;
import org.dei.perla.parser.statements.Statement;
import org.dei.perla.sys.query.llqruntime.datastructures.Record;
import org.dei.perla.utils.logger.LogRecord.Type;
import org.dei.perla.utils.logger.LogRecord.Verbosity;
import org.dei.perla.utils.messages.AdapterFpcMessage;
import org.dei.perla.utils.pipe.Pipe;
import org.dei.perla.utils.waiter.Waitable;
import org.dei.perla.utils.waiter.Waiter;

/**
 * Questa classe astrae un dispositivo fisico connesso al sistema.
 * Si occupa di tradurre e comunicare le richeste verso il
 * dispositivo e le risposte ricevute.
 */
public abstract class FunctionalityProxyComponent extends LoggableComponent
    implements Startable {

  /**
   * La struttura dati contenenente le altre strutture dati necessarie al
       funzionamento della classe.
   */
  private final FPCDataStructure pFPCDataStructure;
```

```
/**
 * Pipe per dati in arrivo dal dispositivo
 */
private final Pipe<AdapterFpcMessage> inputPipe;

/**
 * Waiter dei dati in arrivo dal dispositivo
 */
private final Waiter<Pipe<AdapterFpcMessage>> inputPipeWaiter;

/**
 * Gruppo di pipe per i dati in uscita (un FPC puo' servire piu' DataCollectors)
 */
private final ArrayList<Pipe<Record>> outputPipeArrayList;


/**
 * Inner class che gestice la ricezione dei dati dal dispositivo astratto
 */
private class PipeReader implements Runnable {

  public void run() {

    Waitable<Pipe<AdapterFpcMessage>> waitable;
    Pipe<AdapterFpcMessage> pipe;
    AdapterFpcMessage message;

    while(true) {
      waitable = inputPipeWaiter.waitNext();
      pipe = waitable.getParentComponent();
      message = pipe.dequeue();

      if (message != null) {
        readMessageFromAdapter(message.getPayload());
        /**/
      } else {
        inputPipeWaiter.removeWaitable(waitable);
        putNullOnOutputPipeArrayList();
        pipe.stop();
        pipeClosedCallBack(pipe);
        break;
      }
    }
    loggerNotify(Type.Information, getName() + "␣terminated!", Verbosity.Low);
  }

}

/**
 * Indica se il componente Ãš attualmente avviato
 */
protected boolean started;


/**
 * Metodo per inizializzare l'FPC. Viene eseguito nel costruttore
 */
protected abstract void initDefault();


/**
 * Metodo che implementa il comportamento da seguire nel caso la Pipe di
     comunicazione con il dispositivo fisico
 * sia stata chiusa (sia manualmente, sia in caso di errore)
 * @param pipe
 */
protected abstract void pipeClosedCallBack(Pipe<AdapterFpcMessage> pipe);

/**
 * Invia null sulle pipe verso i DataCollector per segnalarne la chiusura
 */
synchronized final protected void putNullOnOutputPipeArrayList(){
  for(Pipe<?> p: outputPipeArrayList)
    p.enqueue(null);
}

/**
 * Converte uno stream di byte in un record (interpreta)
 * Viene invocato quando l'FPC riceve un messaggio dai livelli sottostanti
 * */
protected abstract void readMessageFromAdapter(byte[] payload);

/**
 * Richiede all'FPC una nuova pipe (da usare quando ci si aggancia un nuovo
     DataCollector)
 */
synchronized final public Pipe<Record> getNewOutputPipe(){
  Pipe<Record> pipe = new Pipe<Record>(this.getName() + "_PIPE_" + (
      outputPipeArrayList.size()+1));
  pipe.start();
  outputPipeArrayList.add(pipe);
  return pipe;
}

/**
 * Costruttore
 * @param parName il nome da dare a questo FPC (e.g: "FPC per i sensori tipo A")
 * @param inputPipe la pipe per scambiare i dati con il dispotivo
 */
public FunctionalityProxyComponent(String parName, Pipe<AdapterFpcMessage>
    inputPipe, FPCDataStructure parDataStructure) {
  super(parName);
  this.started = false;
  this.inputPipe = inputPipe;
  this.inputPipeWaiter = new Waiter<Pipe<AdapterFpcMessage>>();
  this.inputPipeWaiter.addWaitable(inputPipe.getWaitable());
  this.outputPipeArrayList = new ArrayList<Pipe<Record>>();
  this.pFPCDataStructure = parDataStructure;
  this.initDefault();
}
```

```
/**
 * Restituisce TRUE se l'FPC Ãš attivo e funzionante, FALSE altrimenti
 */
synchronized final public boolean isStarted() {
  return started;
}


/**
 * Permette di avviare l'FPC
 */
synchronized public void start() {
  inputPipeWaiter.start();
  new Thread(new PipeReader()).start();
  started = true;
  loggerNotify(Type.Information, getName() + "␣has␣just␣started␣its␣activity",
      Verbosity.Low);

}

/**
 * Permette di stoppare l'FPC
 */
synchronized public void stop() {
  inputPipe.enqueue(null);
  started = false;
  loggerNotify(Type.Information, getName() + "␣has␣just␣stopped␣its␣activity",
      Verbosity.Low);
}

/**
 * Setta il tempo di sampling
 */
public abstract boolean setSamplingTime(long parTime,
    OnUnsupportedSampleRateOptions parBehaviour);

/**
 * Registra una nuova query da eseguire sull'FPC
 * @param parAttrbiutesNames lista contenente i nomi degli attributi richiesti
     dalla query
 * @return TRUE se ok, FALSE se errore
 */
public abstract boolean registerQuery(Statement parStatement);

/**
 * Quando una query termina va de-registrata dall'FPC
 * @return TRUE se ok, FALSE altrimenti
 */
public abstract boolean unRegisterQuery();


/**
 * Restituisce l'elenco di attributi che l'FPC Ãš in grado di campionare (ovvero,
      in altri termini, gli attributi che sta wrappando)
 * @deprecated E' un metodo che va' sicuramente eliminato. Al momento viene
     utilizzato solo dalla prima versione del Registry di Arthur.
```

```
   *
   */
  public abstract ArrayList<NodeLogicalObjectAttribute> getAttributeList();

  /**
   * Normalmente il flusso di record parte dall'FPC e attraverso le pipe arriva a
       destinazione in modo indipendente
   * da quest'ultima (si veda la classe pipe.java). Tuttavia e' talvolta necessario
        forzare il campionamento: caso tipico della
   * clausola REFRESH.
   *
   */
  public abstract ArrayList<Record> forceSampling();
}
```

FPCDataStructure class

```
package org.dei.perla.sys.device.fpc;

import org.dei.perla.parser.expressions.Constant;
import org.dei.perla.utils.dataconverter.DataConverter;


/**
 * Questa classe contiene tutte le strutture dati necessarie al funzionamento dell'
     FPC
 * Non e' stata inserita alcuna funzione complementare alla getAttributeByName, in
     quanto si
 * presuppone che il valore degli attributi contenuti nelle strutture dati sia
     impostato
 * esclusivamente dalla funzione deserializeData, la quale deserializza lo stream
     di Byte
 * in arrivo dal dispositivo fisico e modifica i valori delle strutture
     aggiornandone il contenuto
 * con i dati attuali.
 * TODO: Valutare se l'assunzione che una setAttributeByName non serva sia corretta
 */
public abstract class FPCDataStructure {

  /**
   * Metodo da utilizzarsi per recuperare il valore di un singolo attributo
   * a partire dal nome di questo. Nel caso il nome dell'attributo non appartenga
   * all'FPC verra' restituito null.
   * @param parName nome dell'attributo richiesto
   * @return Valore dell'attributo richiesto. Viene restituito null nel caso l'FPC
   * non contenga l'attributo richiesto
   */
  public abstract Constant getAttributeByName(String parName);

  /**
   * Questo metodo consente di popolare una delle strutture dati presenti nella
       classe
   * a partire dallo stream di dati ricevuto dal dispositivo fisico di cui l'FPC e'
        l'astrazione
   * @param parStructureIdentifier identificatore della classe da popolare
   * @param parRawDataArray array contenente i dati con cui caricare la classe
   */
  public void deserializeData(String parStructureIdentifier, Byte[] parRawDataArray
      ) {
    Object structure = this.getDataStructure(parStructureIdentifier);

    if (structure == null) {
      throw new IllegalArgumentException("La␣struttura␣dati␣" +
          parStructureIdentifier +
      "␣non␣e'␣stata␣trovata!");
    }
    DataConverter.fromByteArray(structure, parRawDataArray);
  }

  /**
   * Questo metodo permette di convertire una struttura dati in un array di Byte
       utilizzabile
   * per l'invio su rete
   * @param parStructureIdentifier identificatore della struttura dati da
       serializzare
   * @return array di byte corrispondente alla serializzazione della classe
   */
  public Byte[] serializeData(String parStructureIdentifier) throws
      IllegalArgumentException {
    Object structure = this.getDataStructure(parStructureIdentifier);

    if (structure == null) {
      throw new IllegalArgumentException("La␣struttura␣dati␣" +
          parStructureIdentifier +
          "␣non␣e'␣stata␣trovata!");
    }
    return DataConverter.toByteArray(structure);
  }

  /**
   * Metodo per il recupero di una struttura dati memorizzata all'interno della
       classe.
   * Puo' essere implementato come una semplice tabella di lookup. L'identificatore
        puo'
   * essere il nome stesso della struttura dati da recuperare
   * @param parStructureIdentifier identificatore della struttura dati da
       recuperare
   * @return struttura dati richiesta. Nel caso l'identificatore non corrisponda a
       nessuna
   * struttura dati presente nella classe deve essere restituito null
   */
  protected abstract Object getDataStructure(String parStructureIdentifier);
}
```

SamplingData class

```
package org.dei.perla.sys.query.llqruntime.datastructures;

import java.util.ArrayList;

import org.dei.perla.parser.clauses.OnUnsupportedSampleRateOptions;
import org.dei.perla.parser.clauses.RefreshClause;
```

```
/**
 * Questa classe rappresenta tutte le informazioni sul sampling (TABELLA, REFRESH,
     UNSUPPORTEDSAMPLERATE, etc)
 * @author Camplani, Fortunato, Marelli, Rota et al.
 */
public class SamplingData {

  /**
   * ArrayList contenente i singoli record della tabella
   */
  private ArrayList<FrequencyRecord> pRecords;

  /**
   * Comportamento da tenere nel caso in cui il dispositivo non supporta una certa
       velocita' di campionamento
   */
  private OnUnsupportedSampleRateOptions pBehaviour;

  /**
   * Campo per la clausola REFRESH
   */
  private RefreshClause pRefreshClause;


  /**
   * Costruttore della tabella, inizializzandola con giÃ  un insieme di record. Di
       tale insieme vengono tenuti
   * solo i record validi (ovvero quelli che sono corretti). Se l'insieme Ãš vuoto
       si ricade nel costruttore banale
   * @param parArrayList l'insieme di record
   */
  public SamplingData(ArrayList<FrequencyRecord> parArrayList,
      OnUnsupportedSampleRateOptions parBehaviour, RefreshClause parRefresh) {

    this.pBehaviour = parBehaviour;
    this.pRefreshClause = parRefresh;
    pRecords = new ArrayList<FrequencyRecord>();

    if (parArrayList == null) return; //anche se io lancerei l'eccezione data l'
        importanza;

    for (FrequencyRecord tRecord : parArrayList) {

      if (checkRecord(tRecord)) {

        this.pRecords.add(tRecord);
      }
    }
  }

  /**
   * @return Il comportamento in caso di frequenza non supportata dalla FPC
   */
  public OnUnsupportedSampleRateOptions getBehaviour() {
    return this.pBehaviour;
  }

  /**
   * Metodo helper privato per il controllo della validitÃ  di un record
   * @param parRecordToCheck il record da controllare
   * @return true se Ãš valido, false altrimenti
   */
  private boolean checkRecord(FrequencyRecord parRecordToCheck) {

    //TODO Eccezione

    if (parRecordToCheck.getFrequency().getValue()>=0) {

      return true;

    } else {

      System.out.println("***␣Record␣non␣inserito␣in␣tabella␣(Errore␣nella␣
          frequenza:):␣"+parRecordToCheck.toString()+"␣***\n");
      return false;
    }
  }

  /**
   * @return la clausola REFRESH per questo sampling
   */
  public RefreshClause getRefreshClause() {
    return this.pRefreshClause;
  }

  /**
   * Ritorna una copia (per sicurezza) del contenuto della tabella
   */
  public ArrayList<FrequencyRecord> getSamplingTableContent() {
    return this.pRecords;
  }


  /**
   * Metodo per stampare a video il contenuto della tabella
   */
  public String toString() {

    String tString = new String();

    for (FrequencyRecord tRecord : pRecords) {

      tString += tRecord.getFirstOperand().toString()+"|"+
            tRecord.getFrequency().toString()+"\n";

    }

    return tString;
  }

}
```

FrequencyRecord class

```
package org.dei.perla.sys.query.llqruntime.datastructures;

import org.dei.perla.parser.expressions.*;
import org.dei.perla.parser.util.Duration;

/**
 * Classe che rappresenta un singolo record della tabella con i valori di
     campionamento
 * @author Camplani, Fortunato, Marelli, Rota et al.
 */
public class FrequencyRecord {

  /**
   * Campi del singolo record
   * |NODO_DA_TESTARE|CONDIZIONE_DI_CONFRONTO|FREQUENZA_SAMPLING|
   */
  private Node pFirstNode;
  private Duration pFrequency;

  /**
   * Costruttore nel caso il confronto sia sempre fatto con TRUE
   * @param parFirst L'espressione che deve verificarsi
   * @param parDuration la frequenza di campionamento associata al verificarsi di
       parFirst
   */
  public FrequencyRecord(Node parFirst, Duration parDuration) {

    this.pFirstNode = parFirst;
    this.pFrequency = parDuration;

  }

  /**
   * @return il primo field del record (espressione da verificare)
   */
  public Node getFirstOperand() {

    return this.pFirstNode;
  }

  /**
   * @return l'oggetto Duration associato al verificarsi del record
   */
  public Duration getFrequency() {

    return this.pFrequency;

  }

  /**
```

```
   * Setter del nodo da verificare
   * @param parNode il nodo da verificare
   */
  public void setFirstNode(Node parNode) {

    this.pFirstNode = parNode;

  }

  /**
   * Restituisce la stringa che rappresenta il contenuto del record
   */
  public String toString() {

    return pFirstNode.toString()+"|"+pFrequency.getValue()+pFrequency.getTimeUnit()
        .toString();
  }

}
```

SamplingData class

```
package org.dei.perla.sys.query.llqruntime.datastructures;

import java.util.ArrayList;

import org.dei.perla.parser.clauses.OnUnsupportedSampleRateOptions;
import org.dei.perla.parser.clauses.RefreshClause;


/**
 * Questa classe rappresenta tutte le informazioni sul sampling (TABELLA, REFRESH,
     UNSUPPORTEDSAMPLERATE, etc)
 * @author Camplani, Fortunato, Marelli, Rota et al.
 */
public class SamplingData {

  /**
   * ArrayList contenente i singoli record della tabella
   */
  private ArrayList<FrequencyRecord> pRecords;

  /**
   * Comportamento da tenere nel caso in cui il dispositivo non supporta una certa
       velocita' di campionamento
   */
  private OnUnsupportedSampleRateOptions pBehaviour;

  /**
   * Campo per la clausola REFRESH
   */
  private RefreshClause pRefreshClause;


  /**
 * Costruttore della tabella, inizializzandola con giÃ  un insieme di record. Di
     tale insieme vengono tenuti
 * solo i record validi (ovvero quelli che sono corretti). Se l'insieme Ãš vuoto
     si ricade nel costruttore banale
 * @param parArrayList l'insieme di record
 */
public SamplingData(ArrayList<FrequencyRecord> parArrayList,
    OnUnsupportedSampleRateOptions parBehaviour, RefreshClause parRefresh) {

  this.pBehaviour = parBehaviour;
  this.pRefreshClause = parRefresh;
  pRecords = new ArrayList<FrequencyRecord>();

  if (parArrayList == null) return; //anche se io lancerei l'eccezione data l'
      importanza;

  for (FrequencyRecord tRecord : parArrayList) {

    if (checkRecord(tRecord)) {

      this.pRecords.add(tRecord);
    }
  }
}

/**
 * @return Il comportamento in caso di frequenza non supportata dalla FPC
 */
public OnUnsupportedSampleRateOptions getBehaviour() {

  return this.pBehaviour;
}

/**
 * Metodo helper privato per il controllo della validitÃ  di un record
 * @param parRecordToCheck il record da controllare
 * @return true se Ãš valido, false altrimenti
 */
private boolean checkRecord(FrequencyRecord parRecordToCheck) {

  //TODO Eccezione

  if (parRecordToCheck.getFrequency().getValue()>=0) {

    return true;

  } else {

    System.out.println("***␣Record␣non␣inserito␣in␣tabella␣(Errore␣nella␣
        frequenza:):␣"+parRecordToCheck.toString()+"␣***\n");
    return false;
  }
}

/**
 * @return la clausola REFRESH per questo sampling
```

```
   */
  public RefreshClause getRefreshClause() {
    return this.pRefreshClause;
  }

  /**
   * Ritorna una copia (per sicurezza) del contenuto della tabella
   */
  public ArrayList<FrequencyRecord> getSamplingTableContent() {
    return this.pRecords;
  }


  /**
   * Metodo per stampare a video il contenuto della tabella
   */
  public String toString() {

    String tString = new String();

    for (FrequencyRecord tRecord : pRecords) {

      tString += tRecord.getFirstOperand().toString()+"|"+
            tRecord.getFrequency().toString()+"\n";

    }

    return tString;

  }

}
```

LLStack class

```
package org.dei.perla.sys.query.llqruntime.datacollection;

/**
 * Rappresenta la coppia DataCollector - Executor
 */
public class LLStack {

  private DataCollector pDataCollector;
  private Object pExecutor;

  public LLStack(DataCollector parDataCollector, Object parExecutor) {
    this.pDataCollector = parDataCollector;
    this.pExecutor = parExecutor;
  }

  public DataCollector getDataCollector() {
    return this.pDataCollector;
  }

  public Object getLLExecutor() {
    return this.pExecutor;
  }

  public void start() {
    this.pDataCollector.start();
    //this.pExecutor.start();
  }

}
```

QueryInformation class

```
package org.dei.perla.sys.query.llqruntime;

import org.dei.perla.parser.statements.Statement;

public abstract class QueryInformation {


  /**
   * Lo statement di cui si rappresentano le info
   */
  private Statement pStatement;


  /**
   * @return Lo statement di cui si rappresentano le info
   */
  public Statement getRepresentedStatement() {
    return this.pStatement;
  }

  /**
   * Costruttore
   * @param parStatement Lo statement di cui si rappresenteranno le informazion
   */
  public QueryInformation(Statement parStatement) {
    this.pStatement = parStatement;
  }

}
```

LLDataCollectionInfo class

```
package org.dei.perla.sys.query.llqruntime.datacollection;

import org.dei.perla.parser.expressions.Node;
import org.dei.perla.parser.statements.Statement;
import org.dei.perla.sys.device.fpc.*;
import org.dei.perla.sys.query.llqruntime.QueryInformation;
import org.dei.perla.sys.query.llqruntime.datastructures.*;


/**
 * Questa classe wrappa tutte le informazione necessarie per effettuare
     correttamente il
 * sampling dei dati dal sistema pervasive
 * @author Schreiber, Camplani, Fortunato, Marelli, Rota et. al.
 */
public class LLDataCollectionInfo extends QueryInformation {

  /**
   * L'oggetto contenente le tempistiche sul campionamento (REFRESH, IF-EVERY-ELSE,
       UNSUPPORTED SAMPLE RATE)
   */
  private SamplingData pSampData;

  /**
   * L'oggetto Node che rappresenta la clusola WHERE
   */
  private Node pWhere;

  /**
   * Il buffer in cui inserire i dati appena raccolti (e filtrati dalla WHERE)
   */
  private Buffer pLocalBuffer;

  /**
   * L'FPC da cui i dati da campionare devono provenire
   */
  private FunctionalityProxyComponent pFPC;

  /**
   * Costruttore
   * @param parStatement Il LLStatemente di cui si stanno rappresentando le info
   * @param parFPC L'FPC da cui i dati da campionare devono provenire
   * @param parSampData L'oggetto contenente le tempistiche sul campionamento (
       REFRESH, IF-EVERY-ELSE, UNSUPPORTED SAMPLE RATE)
   * @param parWhere L'oggetto Node che rappresenta la clusola WHERE
   * @param parBuffer Il buffer in cui inserire i dati appena raccolti (e filtrati
       dalla WHERE)
   */
  public LLDataCollectionInfo(Statement parStatement, FunctionalityProxyComponent
      parFPC, SamplingData parSampData, Node parWhere, Buffer parBuffer) {
    super(parStatement);
    this.pSampData = parSampData;
    this.pWhere = parWhere;
    this.pLocalBuffer = parBuffer;
    this.pFPC = parFPC;
  }

  /**
   * @return Il buffer in cui inserire i dati appena raccolti (e filtrati dalla
       WHERE)
   */
  public Buffer getLocalBuffer() {
    return this.pLocalBuffer;
  }

  /**
   * @return L'FPC da cui i dati da campionare devono provenire
   */
  public FunctionalityProxyComponent getAssociatedFPC() {
    return this.pFPC;
  }

  /**
   * @return L'oggetto contenente le tempistiche sul campionamento (REFRESH, IF-
       EVERY-ELSE, UNSUPPORTED SAMPLE RATE)
   */
  public SamplingData getSamplingData() {
    return this.pSampData;
  }

  /**
   * @return L'oggetto Node che rappresenta la clusola WHERE
   */
  public Node getWhereCondition() {
    return this.pWhere;
  }


}
```

Buffer class

```
package org.dei.perla.sys.query.llqruntime.datastructures;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

import org.dei.perla.parser.expressions.ConstantTimestamp;

public class Buffer {

  private ArrayList<Record> pRecordSet;

  /**
   * Costruisce l'oggetto Buffer, inizializzando il buffer interno con 0 record
   */
  public Buffer(){

    pRecordSet = new ArrayList<Record>();

  }

  /**
   * Costruisce l'oggetto Buffer, inizializzando il buffer interno con spazio per
       tSize record
   * @param tSize Dimensione iniziale del buffer
   */
  public Buffer(int tSize) {

    pRecordSet = new ArrayList<Record>(tSize);
  }
```

```
/**
 * Costruisce l'oggetto Buffer, inizializzando il buffer interno con un set di
     Record
 * @param parRecordSet Set di record
 */
public Buffer(ArrayList<Record> parRecordSet){

  pRecordSet = new ArrayList<Record>(parRecordSet);

}

/**
 * Aggiunge un record al buffer
 * @param parRecord
 * @return true se l'oggetto Ãš stato aggiungto correttamente, false altrimenti
 */
public boolean addRecord (Record parRecord){

  return pRecordSet.add(parRecord);

}

/**
 * Elimina un record dal buffer
 * @param parRecord Il record da eliminare
 * @return true se il record Ãš stato eliminato correttamente, false altrimenti
 */
public boolean delRecord (Record parRecord){

  return pRecordSet.remove(parRecord);

}

/**
 *  Ritorna l'indice del record avente il timestamp specificato
 *  ritorna -1 se il record non Ãš stato trovato
 */
private int getIndexByTimestamp(Timestamp parTimestamp){

  Record tRecord;
    Iterator<Record> tIteratorRecordSet = pRecordSet.iterator();
    //Iterator<Constant> tIteratorConstant;
    ConstantTimestamp tFieldTimestamp;
    //Record tRecord;

  while(tIteratorRecordSet.hasNext()){
    tRecord = (Record)tIteratorRecordSet.next();

    tFieldTimestamp = tRecord.getTimestamp();

    if(tFieldTimestamp.getValueTimestamp() == parTimestamp.getTime())
      return pRecordSet.indexOf(tRecord);

  }
  return -1;
}
```

```
/**
 * Ritorna l'iteratore di una finestra di record
 * @param parInit indice iniziale della finestra
 * @param parEnd  indice finale della finestra
 * @return Iterator
 */
private Iterator <Record> recordsWindowIteratorByIndexes(int parInit, int parEnd)
    {
  ArrayList <Record> tRecordWindowIterator;

  if (parInit>=0 && parEnd < pRecordSet.size() && parInit <= parEnd ){
    tRecordWindowIterator = new ArrayList<Record>(pRecordSet.subList(parInit,
        parEnd));
    return tRecordWindowIterator.iterator();
  }
  else
    return null;
}

/**
 *  Ritorna il record di indice specificato
 */
private Record recordByIndex(int parIndex){

  if (parIndex >=0 && parIndex < pRecordSet.size())
    return (Record)pRecordSet.get(parIndex);
  else
    return null;
}

public Iterator<Record> getIterator(Timestamp parTimestamp, long parDelta){
  int tIndexInit = getIndexByTimestamp(parTimestamp);
  int tIndexEnd  = getIndexByTimestamp(new Timestamp(parTimestamp.getTime() +
      parDelta));

  return recordsWindowIteratorByIndexes ( tIndexInit, tIndexEnd+1);
}

public Iterator<Record> getIterator (Timestamp parTimestamp, int parNumRecord){
  int tIndexInit = getIndexByTimestamp(parTimestamp);

  return recordsWindowIteratorByIndexes(tIndexInit, tIndexInit + parNumRecord);

}

/**
 * Ritorna il contenuto del buffer
 */
public ArrayList<Record> getBuffer() {

  return this.pRecordSet;
}

/**
 * Restituisce la stringa che raccoglie il contenuto del buffer
```

```
   */
  public String toString(){

    String tString = "";
    Iterator<Record> tIterator = pRecordSet.iterator();

    while(tIterator.hasNext())
      tString = tString + ((Record)tIterator.next()).toString() + "\n";

    return tString;
  }

}
```

ActiveSession class

```
package org.dei.perla.sys.environments;

import java.util.*;

/**
 * Questa classe rappresenta una sessione di PerLa attualmente attiva
 */
public class ActiveSession {

  /**
   * Elenco degli environment attivi
   */
  private static ArrayList<Environment> pEnvsArray = new ArrayList<Environment>();

  /**
   * Aggiunge (e attiva) una query, gia' parsata e pronta per essere eseguita
   * @param parEnvironment
   * @return TRUE aggiunta correttamente e fatta partire, FALSE altrimenti
   */
  public static boolean addToSession(Environment parEnvironment) {
    //parEnvironment.start(); //togliere i commenti deve partire appena aggiunto
    return pEnvsArray.add(parEnvironment);
  }

  /**
   * @return L'<code>Iterator</code> per "sfogliare" gli environments attivi
   */
  public static Iterator<Environment> getActiveEnvironments() {
    return pEnvsArray.iterator();
  }

  /**
   * @return Il numero di Statement attualmente attivi
   */
  public static int getEnvironmentsNumber() {
    return pEnvsArray.size();
  }

  /**
   * @return L'elenco completo degli environments attivi
   */
  public static ArrayList<Environment> getEnvironments() {
    return pEnvsArray;
  }
}
```

Environment class

```
package org.dei.perla.sys.environments;

import org.dei.perla.component.Startable;
import org.dei.perla.parser.statements.Statement;

public abstract class Environment implements Startable {

  /**
   * @return Ogni sottoclasse di Environment deve restituire la Query rappresentata
   * nel modo che ritiene piu' opportuno
   */
  public abstract Statement getRepresentedQuery();

}
```

LLEnvironment class

```
package org.dei.perla.sys.environments;

import java.util.*;

import org.dei.perla.component.Startable;
import org.dei.perla.parser.statements.Statement;
import org.dei.perla.sys.device.fpc.FunctionalityProxyComponent;
import org.dei.perla.sys.query.llqruntime.datacollection.LLStack;

public class LLEnvironment extends Environment implements Startable{

  private ArrayList<LLStack> pStacks;

  public LLEnvironment() {
    pStacks = new ArrayList<LLStack>();
  }

  @Override
  public Statement getRepresentedQuery() {
    return this.pStacks.get(0).getDataCollector().getLLDataCollectionInfo().
        getRepresentedStatement();
  }

  public Iterator<FunctionalityProxyComponent> getActiveFPCs() {
    ArrayList<FunctionalityProxyComponent> tArray = new ArrayList<
        FunctionalityProxyComponent>();

    for (LLStack tStack : pStacks) {
      tArray.add(tStack.getDataCollector().getLLDataCollectionInfo().
          getAssociatedFPC());
    }

  return tArray.iterator();
  }

  public ArrayList<LLStack> getActiveLLStacks() {
    return this.pStacks;
  }

  public void addStackToEnvironment(LLStack parStack) {
    this.pStacks.add(parStack);
  }

  public boolean isStarted() {
    return false;
  }


  public void start() {
    for (LLStack tStack : pStacks) {
      tStack.start();
    }

  }

  public void stop() {
    // TODO Auto-generated method stub

  }
}
```

QueryAnalyzer class

```
package org.dei.perla.sys.environments;

import java.util.ArrayList;

import org.dei.perla.component.*;
import org.dei.perla.parser.clauses.*;
import org.dei.perla.parser.expressions.*;
import org.dei.perla.parser.statements.*;
import org.dei.perla.parser.util.Duration;
import org.dei.perla.sys.device.fpc.FunctionalityProxyComponent;
import org.dei.perla.sys.query.llqruntime.datacollection.*;
import org.dei.perla.sys.query.llqruntime.datastructures.*;
import org.dei.perla.sys.query.llqruntime.execution.LLExecutionInfo;
import org.dei.perla.sys.query.sqruntime.SetInformation;
import org.dei.perla.sys.registry.MapRegistry;
import org.dei.perla.sys.registry.Registry;
import org.dei.perla.utils.logger.LogRecord.*;

/**
 * Questa classe si occupa della creazione di tutte le strutture dati necessarie
     all'esecuzione di una
 * query in PerLa.
 * @author Camplani, Fortunato, Marelli, Rota et al.
 */
public class QueryAnalyzer extends Component {

  /**
   * Costruttore private, per essere singleton
   * @param parName Il nome del componente (QueryAnalyzer) da passare a Component
   */
  private QueryAnalyzer(String parName) {
    super(parName);
  }

  /**
   * Reference all'unica istanza
   * @see costruttore
   */
  private static QueryAnalyzer pInstance = null;

  /**
   * Usato per notificare al sistema cio' che viene compiuto dal QueryAnalyzer
   * @param parMessageType Il tipo del messaggio
   * @param parMessage  Il messaggio
   * @param parVerbosity La verbositÃ  minima del messaggio per essere stampato a
       video
   * @see Logger
   */
  private static void notifyToLogger(Type parMessageType, String parMessage,
      Verbosity parVerbosity) {
      getInstance().loggerNotify(parMessageType, parMessage, parVerbosity);
  }

  /**
   * @return L'unica instanza del QueryAnalyzer
   */
  public static QueryAnalyzer getInstance() {
    if (pInstance == null) {
      pInstance = new QueryAnalyzer("QueryAnalyzer");
      return pInstance;
    }
    else return pInstance;
  }

  /**
   *Registra la query di PerLa nel sistema. Per ogni Query da eseguire e'
       sufficente
   *chiamare questo metodo che crea tutto il necessario e lo mette in opera.
   */
  public static void registerQuery(Query parQuery) {
    notifyToLogger(Type.Information,"Aggiunta␣query␣in␣corso",Verbosity.High);
    createLLStatements(parQuery.getLLStatements());
    createHLStatements(parQuery.getHLStatements());
    createSetParametersEnvironments(parQuery.getSetStatements());
  }

  /**
   * Metodo che si occupa di creare gli <code>Statement</code> di basso livello
   * @param parList Una lista di statement
 */
private static void createLLStatements(LLStatementList parList) {

  //...qualche dichiarazione
  SamplingData tTable;
  Node tWhere;
  LLEnvironment tEnvironment;
  LLDataCollectionInfo tDCInfo;
  LLExecutionInfo tEXInfo;
  Buffer tBuffer;

  //..per ogni statement di basso livello
  for (LLStatement tStatement : parList) {

    //...tutte le cose condivise dai vari LLDataCollectionInfo
    tEnvironment = new LLEnvironment();
    tTable = getFrequenciesTable(tStatement);
    tWhere = getWhereCondition(tStatement);

    for (FunctionalityProxyComponent tFPC : MapRegistry.getInstance().query(
        tStatement.getExecuteIfClause().getCondition())) {

      tBuffer = new Buffer();
      tDCInfo = new LLDataCollectionInfo(tStatement,tFPC,tTable,tWhere,tBuffer);
      tEXInfo = new LLExecutionInfo(tStatement,tBuffer);

      //..creo lo stack di basso livello
      tEnvironment.addStackToEnvironment(new LLStack(new DataCollector(tDCInfo),
          new Object()));
      tFPC.registerQuery(tStatement);
    }


    for (FunctionalityProxyComponent tFPC : Registry.askRegistryForFPCs(
        tStatement)) {
      tFPC.getClass();
      //...per ogni FPC ottenuto dal registry serve un DC, un LLQExec e un LB
      tBuffer = new Buffer();
      tDCInfo = new LLDataCollectionInfo(tStatement,tFPC,tTable,tWhere,tBuffer);
      tEXInfo = new LLExecutionInfo(tStatement,tBuffer);
      tEXInfo.hashCode(); //Rimuovere, serve per togliere warning

      //..creo lo stack di basso livello
      tEnvironment.addStackToEnvironment(new LLStack(new DataCollector(tDCInfo),
          new Object()));
    }

    //...e aggiungo il tutto al contesto
  ActiveSession.addToSession(tEnvironment);
  }
  notifyToLogger(Type.Information,"Terminata␣creazione␣dei␣LL␣Environments",
      Verbosity.Low);
}

/**
 * Costruisce la tabella con le clausole IF-EVERY-ELSE
 */
private static SamplingData getFrequenciesTable(LLStatement parStatement) {

  SamplingClause tClause = parStatement.getSamplingClause();

  if (tClause instanceof SamplingTimeBased) {
    notifyToLogger(Type.Information, "Sampling␣is␣TIME␣based", Verbosity.Low);
    ArrayList<FrequencyRecord> tArrayList = new ArrayList<FrequencyRecord>();
      for (SamplingIfEvery tSampling : ((SamplingTimeBased) tClause).getPeriods()
          ) {
        Duration tDuration = null;

        try {
          tDuration = new Duration(tSampling.getSamplingRate().getResult().
              getValueFloat(),tSampling.getTimeUnit());
            }
        catch (Exception e) {
          notifyToLogger(Type.Error, "Errore␣nel␣recupero␣del␣valore␣della␣
              frequenza", Verbosity.Low);
            }

    tArrayList.add(new FrequencyRecord(tSampling.getCondition(),tDuration));

    }


  return new SamplingData(tArrayList,((SamplingTimeBased) tClause).getOperation()
      ,((SamplingTimeBased) tClause).getRefreshClause());

  }else {
    notifyToLogger(Type.Information, "Sampling␣is␣EVENT␣based,␣no␣sampling␣data␣
        needed", Verbosity.Low);
    return null;
    }
}

/**
 * Metodo interno per il recupero della clausola WHERE
 */
private static Node getWhereCondition(LLStatement parStatement) {
  return parStatement.getSamplingClause().getWhereClause().getCondition();
}


/**
 * Metodo che si occupa di creare gli <code>Statement</code> di alto livello
 * @param parList Una lista di statement
 */
private static void createHLStatements(HLStatementList parList) {

}

/**
 * Metodo che si occupa di creare gli <code>Statement</code> per le Actuation
     Queries
 * @param parList la lista di SetParameters Query
```

```
   */
  private static void createSetParametersEnvironments(SetParametersStatementList
      parList) {
    for (SetParametersStatement tStatement : parList) {
        ActiveSession.addToSession(new SetEnvironment(new SetInformation(tStatement
            ,tStatement.getAttributesToSet(),tStatement.getIDs())));
    }
  }

}
```

ExpressionAnalyzer class

```
package org.dei.perla.sys.environments;

import java.util.ArrayList;
import org.dei.perla.parser.expressions.*;

/**
 * Permette di analizzare le espressioni alla ricerca di campi oggetto logico.
 * @author Camplani, Fortunato, Marelli, Rota et al.
 *
 */
public class ExpressionAnalyzer {

  private ArrayList<NodeLogicalObjectAttribute> tNodeArrayList = new ArrayList<
      NodeLogicalObjectAttribute>();

  /**
   * Costruttore della classe ExpressionAnalyzer
   */
  public ExpressionAnalyzer() {

  }


  /**
   * Metodo helper privato che si occupa di analizzare se un singolo nodo Ãš un
       campo oggetto logico
   * @param parNode Il nodo da analizzare
   */
  private void getFields(Node parNode) {

    if (parNode instanceof NodeOperation) {

      this.getFields(((NodeOperation) parNode).getFirstNode(), ((NodeOperation)
          parNode).getSecondNode());

    }
    else if (parNode instanceof NodeAggregation) {

      this.getFields(((NodeAggregation) parNode).getExpressionToAggregate());

    }
    else if (parNode instanceof NodeBetween) {
```

```
    this.getFields(((NodeBetween) parNode).getFirstNode(), ((NodeBetween) parNode
        ).getSecondNode(), ((NodeBetween) parNode).getThirdNode());

  }
  else if (parNode instanceof NodeFunction) {

    this.getFields(((NodeFunction) parNode).getFunctionPar());

  }
  else if (parNode instanceof NodeLogicalObjectAttribute && !this.tNodeArrayList.
      contains(parNode)) {

    this.tNodeArrayList.add((NodeLogicalObjectAttribute)parNode);

  }

}

/**
 * Metodo helper che si occupa di analizzare un nodo con due "figli" alla ricerca
      di campi
 * @param parFirstNode Il primo nodo
 * @param parSecondNode Il secondo nodo
 */
private void getFields(Node parFirstNode, Node parSecondNode) {

  this.getFields(parFirstNode);
  if (parSecondNode!=null) {this.getFields(parSecondNode);}

}

/**
 * Metodo helper che analizza i Nodi con tre figli (e.g. NodeBetween)
 * @param parFirstNode Il primo nodo
 * @param parSecondNode Il secondo nodo
 * @param parThirdNode Il terzo nodo
 */
private void getFields(Node parFirstNode, Node parSecondNode, Node parThirdNode)
    {

  this.getFields(parFirstNode);
  this.getFields(parSecondNode);
  this.getFields(parThirdNode);

}

/**
 * Ottiene l'arrayList di tutti i "campi oggetto logico" dei nodi contenuti in un
      arraylist
 * @param parArrayListNode l'arrayList contente i nodi
 */
public ArrayList<NodeLogicalObjectAttribute> getFields(ArrayList<Node>
    parArrayListNode) {

  for (Node tNode : parArrayListNode) {
      this.getFields(tNode);

    }
    return this.tNodeArrayList;

  }

  /**
   * Metodo per ottenere l'ArrayList di tutti i "campi oggetto logico" partendo da
       un nodo radice
   * @param parNode Il nodo radice dell'espressione
   * @return ArrayList contenente i "campi oggetto logico" trovati
   */
  public ArrayList<NodeLogicalObjectAttribute> getNodeArrayList(Node parNode) {

    getFields(parNode);
    return this.tNodeArrayList;

  }
}
```

Pipe class

```
/* Copyright (C) 2008  Romolo Camplani, Marco Fortunato, Marco Marelli, Guido Rota,
     Fabio A. Schreiber et al.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
 */
package org.dei.perla.utils.pipe;

import java.util.LinkedList;
import java.util.Queue;

import org.dei.perla.component.Component;
import org.dei.perla.utils.waiter.Waitable;

/**
 * Coda generica di messaggi. La coda gestisce la sincronizzazione tra i produttori
 * (anche piu' di uno) ed il consumatore (sempre e solo uno).
 * Le operazioni di estrazione dalla coda sono non bloccanti (viene ritornato null
     se
 * non ci sono dati in coda) mentre le operazioni di inserimento sono bloccanti (
     nel
 * caso in cui la coda risulti piena).
 */
public class Pipe<E> extends Component {

  /**
   * Lista dei messaggi presenti nella Pipe
   */
  private final Queue<E> pMessageQueue;

  /**
   * Numero massimo di messaggi che possono essere accodati (0 = infinito)
   */
  private final int pMessageQueueMaxLenght;

  /**
   * Oggetto Waitable per notificare ad un eventuale Waiter la presenza di
   * messaggi nella coda
   */
  private final Waitable<Pipe<E>> pWaitable;

  /**
   * Costruttore (costruisce una pipe illimitata)
   */
  public Pipe(String parName) {
    this(parName, 0);
  }

  /**
   * Costruttore
   * @param parMessageQueueLenght Massimo numero di messaggi che la Pipe potrÃ
       contenere (0 = infinito)
   */
  public Pipe(String parName, int parMessageQueueMaxLenght) {
    // Imposta il nome del componente
    super(parName);

    // Memorizza il massimo numero di messaggi che la Pipe potrÃ  contenere
    if(parMessageQueueMaxLenght < 0) {
      parMessageQueueMaxLenght = 0;
    }
    pMessageQueueMaxLenght = parMessageQueueMaxLenght;

    // Istanzia la coda in cui verranno inseriti i messaggi
    pMessageQueue = new LinkedList<E>();

    // Istanzia l'oggetto Waitable
    pWaitable = new Waitable<Pipe<E>>(this);
  }

  /**
   * Avvia il componente pipe
   */
  @Override
  public void start() {
    synchronized (this.pMessageQueue) {
      if(!isStarted()) {
        // Segnala che la pipe e' avviata
      setStarted(true);
    }
  }
}

/**
 * Arresta il componente pipe
 */
@Override
public void stop() {
  synchronized (pMessageQueue) {
    if(isStarted()) {
      // Svuota la pipe
      pMessageQueue.clear();

      // Risveglia eventuali thread in attesa
      pMessageQueue.notifyAll();

      // Resetta il segnale nell'oggetto Waitable
      pWaitable.resetWaiter();

      // Segnala che la pipe non e' piu' avviata
      setStarted(false);
    }
  }
}

/**
 * Accoda un nuovo messaggio alla Pipe.
 * Se la Pipe risulta piena, la chiamata e' bloccante
 */
public void enqueue(E parMessageToEnqueue) {
  synchronized (pMessageQueue) {

    // Verifica che la pipe sia avviata
    if(!this.isStarted())
      return;

    // Se e' prevista una lunghezza massima della coda ..
    if(pMessageQueueMaxLenght > 0) {

      // .. e questa lunghezza e' stata raggiunta ..
      while(pMessageQueue.size() >= pMessageQueueMaxLenght) {

        // .. si mette in attesa
        try {
          pMessageQueue.wait();
        }
        catch(Exception parException) {
          // TODO: mettere il relay di InterruptedException
        }
      }
    }

    // Aggiunge il nuovo messaggio alla coda
    pMessageQueue.add(parMessageToEnqueue);
```

```
      // Segnala agli Waiter in attesa che ci sono messaggi in coda
      pWaitable.notifyWaiter();
    }
  }

  /**
   * Estrae dalla pipe il messaggio meno recente.
   * Se la Pipe e' vuota, la chiamata ritorna null.
   * @return oggetto recuperato dalla Pipe, null se la Pipe e' vuota.
   */
  public E dequeue() {
    E tDequeuedMessage = null;

    synchronized(pMessageQueue) {

      // Verifica che la pipe sia avviata
      if(!isStarted())
        return null;

      // Se la pipe non e' vuota..
      if(!pMessageQueue.isEmpty()) {
        // .. toglie dalla pipe il messaggio meno recente
        tDequeuedMessage = pMessageQueue.poll();

        // .. risveglia eventuali thread in attesa di poter inserire
        pMessageQueue.notifyAll();

        // .. se la pipe si e' svuotata, resetta il segnale nell'oggetto Waitable
        if(pMessageQueue.isEmpty()) {
          pWaitable.resetWaiter();
        }
      }
    }

    // Restituisce il messaggio che e' stato tolto dalla pipe
    return tDequeuedMessage;
  }

  /**
   * Restituisce un riferimento all'oggetto Waitable
   */
  public Waitable<Pipe<E>> getWaitable() {
    return pWaitable;
  }
}
```

Bibliography

[1] Ian F. Akyildiz Tommaso Melodia Kaushik R. Chowdhury. A survey on wireless multimedia sensor networks. *Comput. Netw.,*, 51:921–960, 2007.

[2] M. Fortunato M. Marelli. "Design of a declarative language for pervasive systems". Master's thesis, Politecnico di Milano, 2006.

[3] Prometeo Project Website. http://www.prometeo.polimi.it, 2009.

[4] Google Protocol Buffers Website. http://code.google.com/p/protobuf/, 2010.

[5] F.A. Schreiber R. Camplani M. Fortunato M. Marelli F. Pacifici. "PerLa: A data language for pervasive systems". In *Proc. of the Sixth Annual IEEE International Conference on Pervasive Computing and Communication (Percom 2008)*, pages 282–287, Piscataway, NJ, USA, 2008. IEEE.

[6] PerLa Website. http://perla.dei.polimi.it, 2009.

[7] A. Pierantozzi R. Puricelli S. Vavassori. "PERLA: Functionality Proxy Component.". Technologies for Information Systems project report, 2008.

[8] A. Maesani C. Magni E. Padula. "Low Level Architecture of the PerLa middleware". Technologies for Information Systems project report, Politecnico di Milano, 2009.

[9] Debasis Saha Amitava Mukherjee. Pervasive computing: a paradigm for the 21st century.

[10] Cappiello Cinzia Schreiber A. Fabio. Quality and energy-aware data compression by aggregation in WSN data streams. In *PerCom Workshops*, pages 1–6, 2009.

[11] ART DECO Website. http://artedeco.elet.polimi.it, 2008.

[12] TinyOs Website. http://www.tinyos.com, 2008.

[13] K. Aberer, M. Hauswirth and A. Saleni. "Global Sensor Network". Technical Report LSIR-REPORT-2006-001, School of Computer and Communication Sciences Ecole Polytechnique Federale de Lausanne (EPFL), 2006.

[14] D. Chu, A. Tavakoli, L. Popa and J. Hellerstein. "Enterely declarative sensor network systems". In *Proc. VLDB '06*, pages 1203–1206, 2006.

[15] Website. http://webdoc.siemens.it/cp/sis/press/sword.htm, 2007.

[16] M. Marino. "A proposal for a context-aware extension of PerLa language". Technologies for Information Systems project report, Politecnico di Milano, 2009.

[17] S. Vettor. "Definizione ed implementazione della rappresentazione interna delle espressioni del linguaggio PERLA.". Undergraduate thesis, Politecnico di Milano, 2008.

[18] L. Baresi D. Braga M. Comuzzi F. Pacifici P. Plebani. "A service infrastructure for advanced logistics". In *IW-SOWE '07: 2nd International Workshop on Service Oriented Software Engineering*, pages 47–53, New York, NY, USA, 2007. ACM press.

[19] P. Prazzoli G. Rota. "Parser design for PERLA query language". Technologies for Information Systems project report, Politecnico di Milano, 2007.

[20] A. Perrucci. "Definizione di una struttura ad oggetti per la rappresentazione interna delle interrogazioni del linguaggio PERLA.". Master's thesis, Politecnico di Milano, 2006.