Politecnico di Milano
V Facoltà di Ingegneria

Progetto di Ingegneria Informatica (5cr.)
A.Y. 2008/2009

Project Report:

PerLa: *DataCollector* design and implementation

Project by:
Diego Vigano' (mat. 668589)

Abstract

This project aims at the definition and the implementation of a data collector within PerLa project [3, 11], in order to collect data from the underlying pervasive system and prepare them for further computations at upper levels.

Contents

List of Figures

1 General overview

This chapter presents a general overview on PerLa in order to give the reader a short overview on the whole project.

1.1 About PerLa

During last years, academic interests in Wireless Sensor Networks (WSN) have rapidly grown followed, at the same rate, by the evolution of WSNs from a handful of homogeneous sensors to a set of heterogeneous ones, whose mutual differences lie in complexity, capabilities and architecture. Along with such technology improvements, some difficulties are arising among application developers, who have to manage these differences in order to make the interaction between different protocols and architectures possible. Developers are also trying to provide WSN users with simple, efficient, and optimized interaction models, able to hide the underlying heterogeneity.

Born within ART DECO project [10], PerLa (*PERvasive LAnguage*) project aims at defining a language that can deal with the above mentioned differences, hiding the complexity of the underlying pervasive system to the final user. The first step in this direction has been the analysis of other similar languages in order to discover their limitations. Studies firstly focused on TinyDB, [13] whose merit lies in the abstraction of a set of homogeneous nodes as a database: this has led TinyDB creators to the definition of an SQL-like language to “query” the system itself. In spite of this great idea TinyDB is limited in the way that it is able to deal only with homogeneous systems: PerLa project aims at allowing to query systems composed of sensors belonging to different technologies. Moreover, PerLa has the goal of allowing devices to be added and removed “on the fly”, without affecting the whole system and completely hiding the needed complexity to the final user. This goal is achieved both through the definition of a SQL-like *language* and the implementation of a *middleware*, able to execute queries written in such language, as explained in the following sections. More information about the comparison between PerLa and other languages can be found in [1] and [3]. The rest of the chapter is organized as follows: an introduction to PerLa semantics and a general syntax description is given in Section 1.1.1, then the middleware architecture is briefly summarized to give the reader a view on the current development state of PerLa.

1.1.1 PerLa language

Since PerLa language was firstly designed [1], the objective of reaching the semantics mentioned before 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 was in fact designed trying to support both "standard" queries (i.e. queries written to retrieve some information from the system), 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. 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 1.1.1.3, a complete example of a quite complex PerLa query is shown.

1.1.1.1 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 unbounded table that collects records produced by LLQs or by others HLQs (especially when this data structure is used as intermediary, like 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.

1.1.1.2 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 1.1.1.3 for an example). 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 [1], and allow the user to specify how to manage the sampled data when the selection is performed. Again, in Subsection 1.1.1.3 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 1.2) 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 [2].

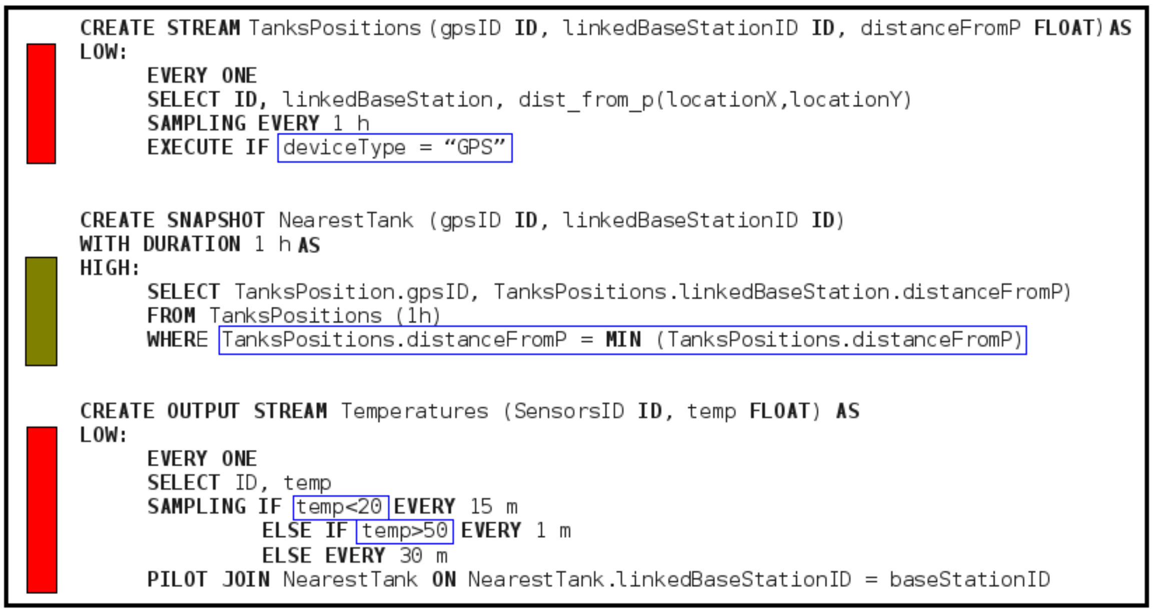
Termination condition section

The termination condition is the last block of an LLQ and allows the user to stop query computation after 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 [1].

1.1.1.3 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 1.1: Query (quite complex) example

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*. The following subsection describes how clauses and expressions are going to be represented within PerLa.

1.1.2 PerLa middleware architecture

PerLa language needs to be fully supported by the middleware: a previous work within the ART DECO project [6] 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.

1.1.2.1 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* [7], 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 [5]: 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 [9], distinguishing between nodes and constants. Parser signals potential syntactic errors and the query is ready to be pushed to the next level when the parsing operation is concluded successfully.

1.1.2.2 Logical Level

The logical level is the most important level in PerLa middleware aiming at accomplishing the following Perla fundamental operations on the pervasive system:

1. dealing with its complexity and heterogeneity
2. computing query results (both of LLQ and HLQ)

The main idea to perform the first operation is to wrap sets of homogeneous sensors into a physical device abstraction called *logical object* (see Figure 1.2). 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

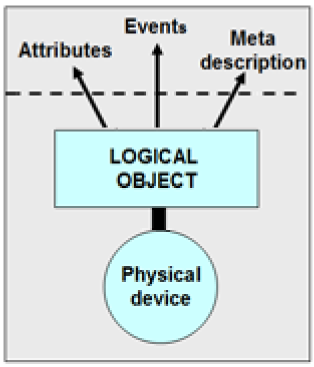


Figure 1.2: Logical object abstraction

With the introduction of the logical object concept, heterogeneity management objective is achieved since pervasive system can now be abstracted as a *group* of logical objects also hiding complexity to the end users.

Let's now focus on the second operation, particularly on LLQ execution. It will certainly require logical objects implementation to include at least 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 Chapter 2 and 3, since the purpose of this project was to actually implement part of the LLQ execution engine.

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 [8], while implementation and integration within PerLa is explained in [4].

1.1.2.3 Device access 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 dialog possible. More information on library architecture can be found in [11].

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.

1.2 Final PerLa overview

Having introduced both the language and 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 1.3) is injected in PerLa.

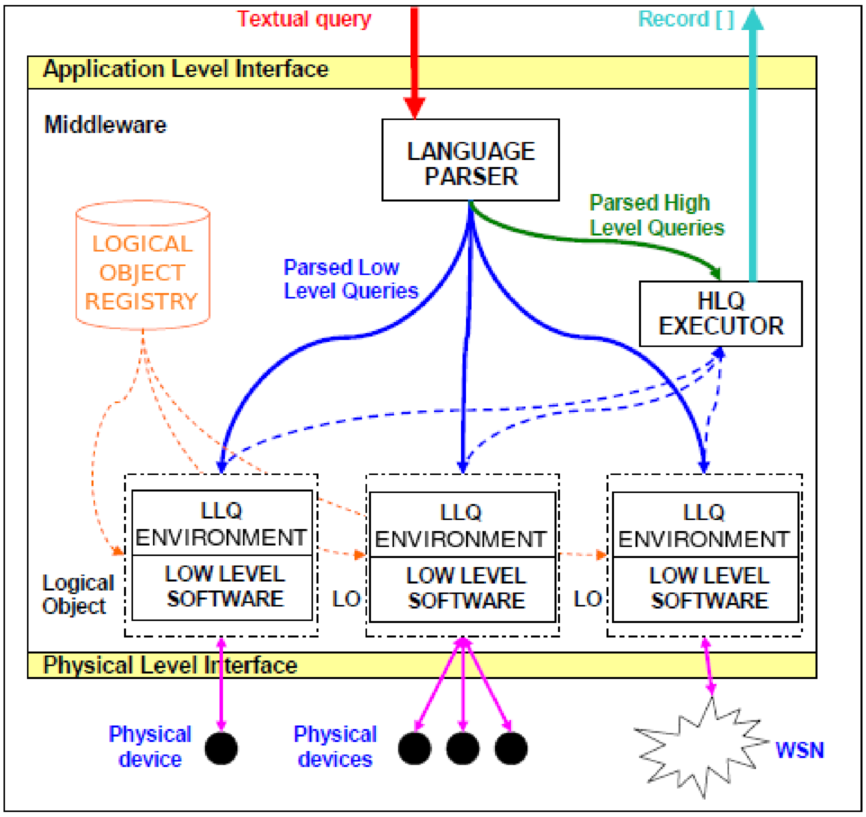


Figure 1.3: PerLa middleware overview

As preliminary passage, the textual query is analyzed by the parser obtaining the Java representation explained in Section 1.1.2.2. LLQs obtained from parser can now be injected in the proper logical objects. As mentioned in Subsection 1.1.1.2 such goal is achieved using a dedicated component. At this point of dissertation something more can be said about this important element: it is called *Logical Object Registry* and 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: 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) 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).

1.2.1 Middleware deployment

It's now worth to briefly analyze where, within a pervasive system, every single component of a PerLa query is executed. The parsing of the query, as well as the evaluation of the set 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 [12], where small sensors called *Acquisition and Elaboration Units* (shortened *AEUs*) will be used to monitor and prevent rockfalls. As can be seen from Figure 1.4, 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 1.4 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 achived on the machine used to monitor the pervasive system, connected to the Gateway using a 5 GHz *TCP/IP* radio bridge.

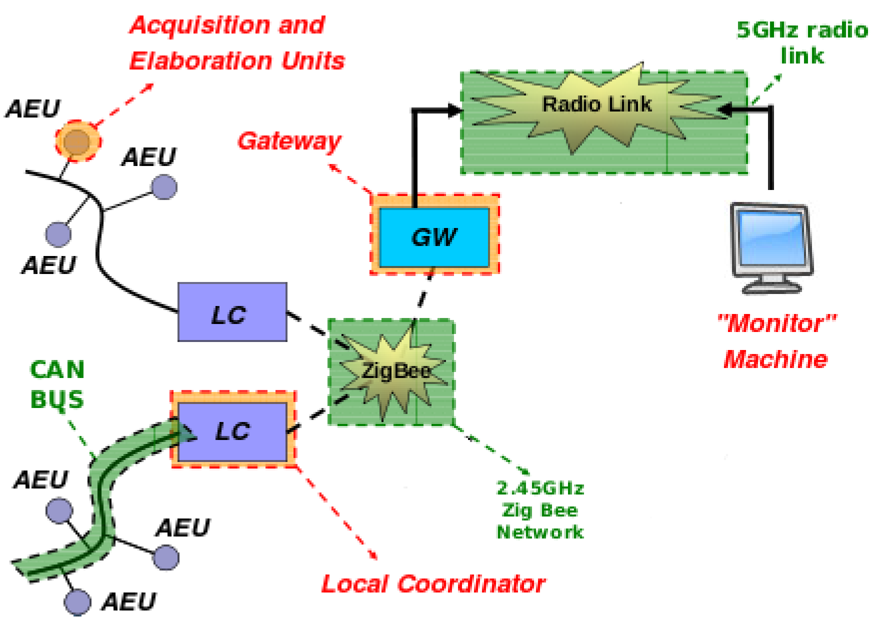


Figure 1.4: A pervasive system example [12]

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 have to be introduced in order to supply such computational capabilities lack. 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 been directly executed on the node.

1.3 About this project

As mentioned in Section 1.1.2.2 this project founds its location within logical level, and particularly in logical objects: while the FPC design has been achieved in [8] and partially implemented in [4], Low Level Query Execution still needed to be designed and implemented, especially for those elements which are responsible to make data exchange possible between FPC and LLQ Environment. The following chapters are focused on these two aspects, that correspond to the steps really undertaken: Chapter 2 will focus on design aspects (essentially explaining how LLQ Execution is expected to be performed) while Chapter 3 focuses on the implementation details of the proposed design.

2 LLQ Execution Design

As mentioned in the previous chapters Low Level Query Execution can be summarized, from a general point of view, as composed of two important parts:

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

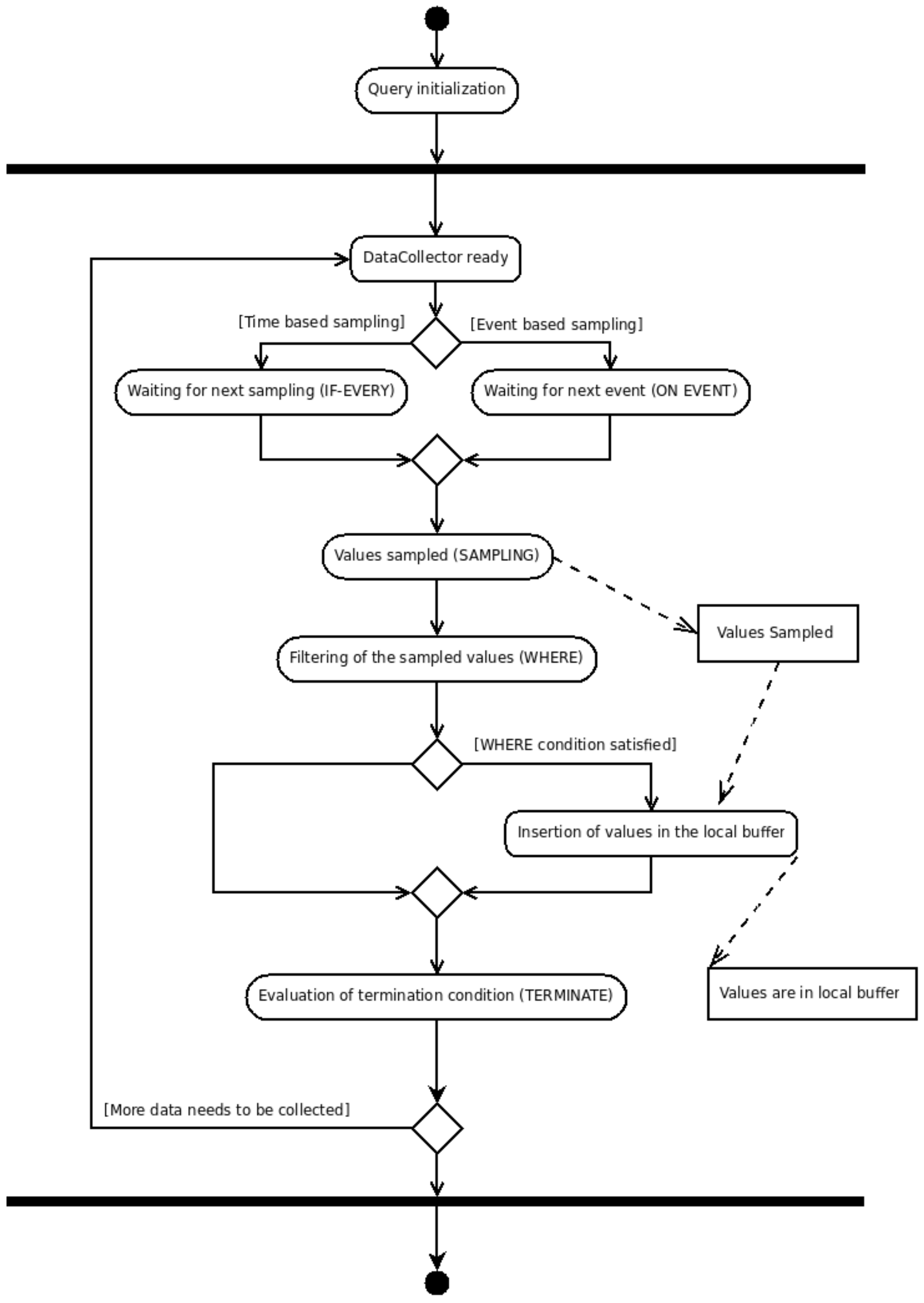
Each part requires a complete design process whose details are presented in the following sections.

2.1 Data collection

First step in the execution of a LLQ is to obtain data: sampling section must in fact be executed before data management section can compute LLQ results. In other terms, local buffer needs to be filled with a certain amount of data (precise number however differs from query to query) before results can be computed. Design phase started studying how this process could be performed, introducing the idea of the *DataCollector* entity, that is charged, as suggested by the name, to the collection of values from the pervasive system. Section 2.1.1 and following focus on this aspect.

2.1.1 DataCollector activity

Data collection activity is quite simple from the design point of view and it is represented in Figure 2.1 using an UML activity diagram:



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

Figure 2.1: Data collection UML activity diagram

The first 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. If sampling condition has changed (e.g. IF-EVERY-ELSE block calls for another frequency) new frequencies are sent to FPC.
- Event based sampling: the event causing sampling is waited for until it occurs.

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 local buffer. 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 2.1 is repeated again.

2.2 Data management

Data management design is currently under review by another project, while a stub design was introduced in [1]. Using such stub as starting point, similarly to the approach used for DataCollector, data management design aims at finding weak and strength point of this component leading to the design of a *Low Level Executor* entity.

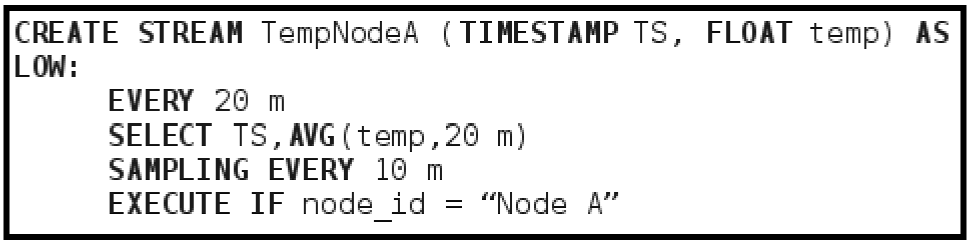
2.3 LLQ Execution design

This section wants to highlight how the aforementioned entities cooperate in order to compute LLQ results. First of all let's focus on DataCollector lifecycle. In the previous subsection, DataCollector has been supposed already created and ready to operate. Actually 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 modelling is carried out at the beginning of its lifecycle during *query initialization* phase; this is represented at the top of the UML diagram in Figure 2.1. Query initialization itself is actually performed by 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 an ad-hoc DataCollector. QueryAnalyzer is, in fact, able to navigate through the received Java structures, searching for terminate conditions, sampling type (time/event based), WHERE clause condition, and produces as output a DataCollector whose activity diagram is fitted for the query analyzed.

It's now possible to explain how LLQ Execution is supposed to operate, using a complete example. Let's suppose that PerLa is operating on a pervasive system composed of four nodes, equipped with the following onboard 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 Figure 2.2, is injected into the system.



```
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"
```

Figure 2.2: First query injected

After the query is analyzed 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 1.1.1.2;

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 2.3.

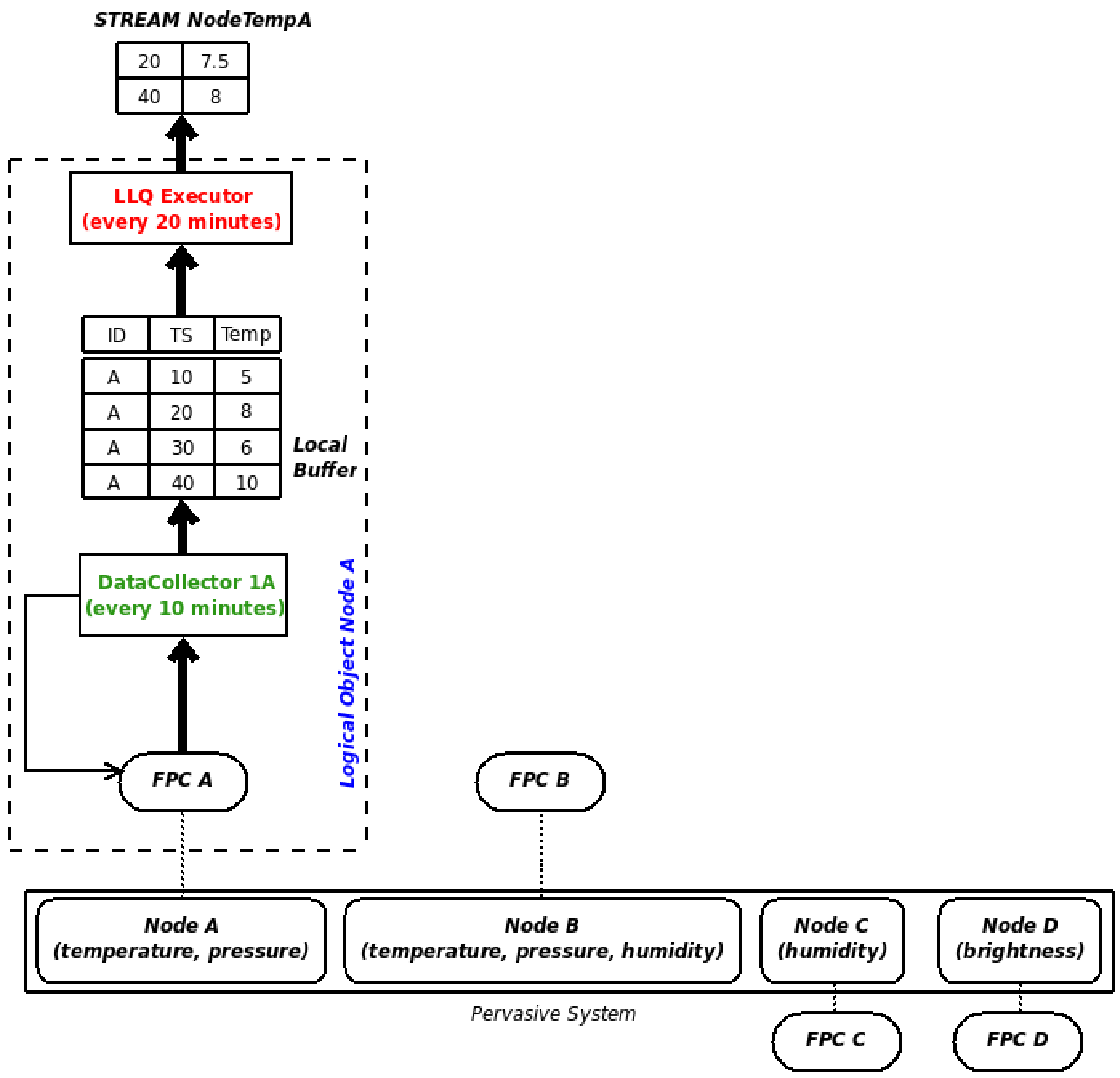
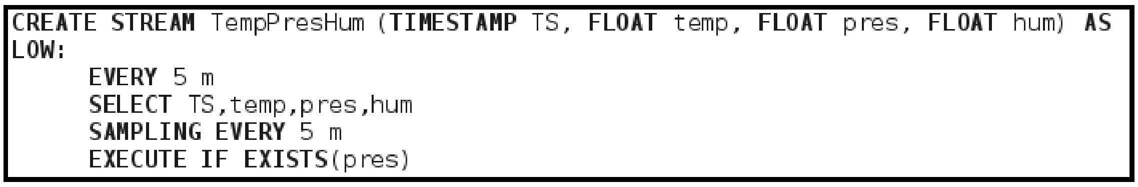


Figure 2.3: LLQ Execution (single query)

This example has 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, exposed in Figure 2.4, is injected into the same system of the previous example.



```
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)
```

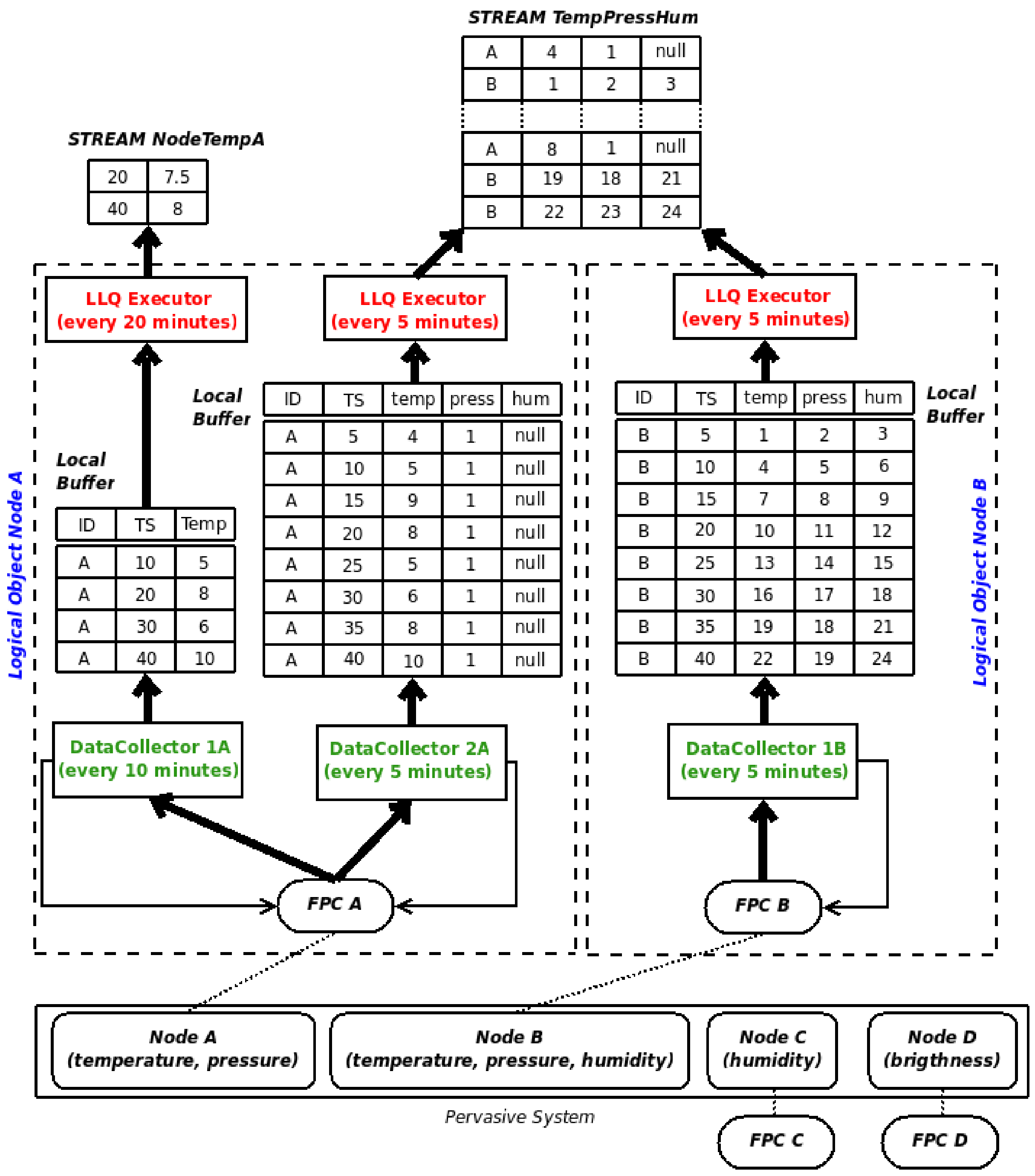
Figure 2.4: Second query injected

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 onboard, 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 query running simultaneously, is reported in Figure 2.5 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
Logical Object Node A
Logical Object Node B
DataCollector 1A (every 10 minutes)
DataCollector 2A (every 5 minutes)
DataCollector 1B (every 5 minutes)
FPC A
FPC B
Node A (temperature, pressure)
Node B (temperature, pressure, humidity)
Node C (humidity)
Node D (brigthness)
Pervasive System
FPC C
FPC D

STREAM TempPressHum

A	4	1	null
B	1	2	3
⋮	⋮	⋮	⋮
A	8	1	null
B	19	18	21
B	22	23	24

STREAM NodeTempA

20	7.5
40	8

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 2.5: 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 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.

2.4 From design to implementation

LLQ execution represents the heart of the middleware and its design required to take into account all the possible interactions among PerLa components: both the blocks still implemented as well as the ones still to be realized have been considered during the design phase: many problems arose in the effort of making “communication” possible between the application level and the device level. A top-down approach was chosen, starting from the simplified scheme exposed in Figure 2.6 which hypothesizes a single and time based simple query running in the system at the moment DataCollector is started, and only one FPC currently associated to it. Since data management part is still to be completed, the collected data are simply appended to the unique local buffer and no more manipulated. Chapter 4 will explain how those limits are intended to be overtaken.

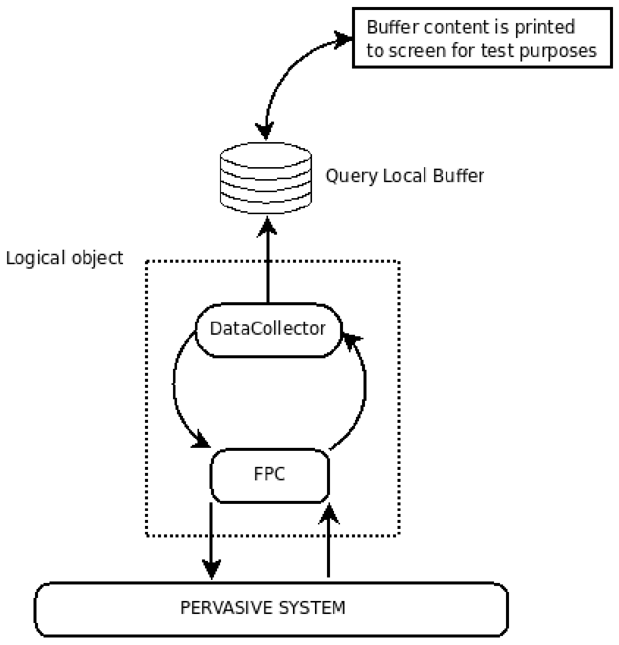


Figure 2.6: Data collection simplified design

3 DataCollector Implementation

In this chapter DataCollector implementation is explained. In the previous chapter terms like “query”, “sampled values” and “structure” have been used in a general fashion and always referring to the functionalities of these components. Section 3.1 is intended to explain what these components actually are, and which data structures are involved.

3.1 Data structures implementation

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

3.1.1 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 [9]), "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 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 3.1. This figure is a summary diagram representing the interaction among these three classes presented before.

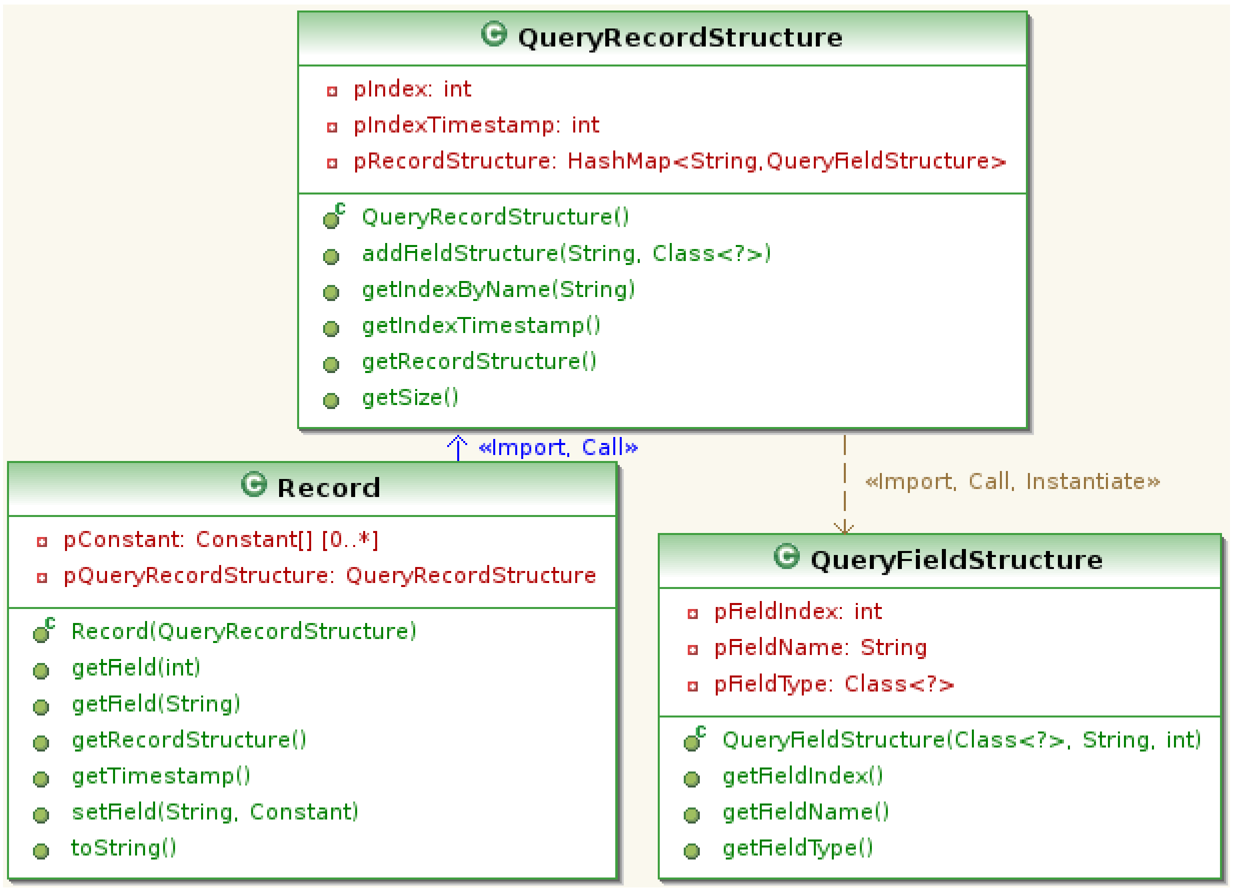


Figure 3.1: Data structures for records UML diagrams

NodeLogicalObjectAttribute class

NodeLogicalObjectAttribute [9] 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 one information known at parsing time (pValue still need 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 3.1.3.

3.1.2 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;
```

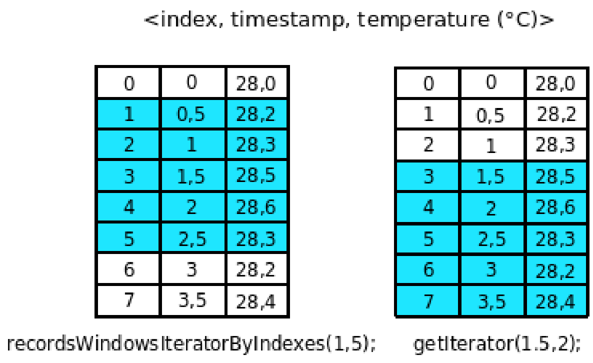
The real importance of buffer class lies in the methods that are provided to navigate through a set of records. The following methods are defined

```
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 3.2 shows the behaviour of these two methods on a buffer taken as example, colouring in light blue the set of records returned.


<index, timestamp, temperature (°C)>
recordsWindowsIteratorByIndexes(1,5);
getIterator(1.5,2);

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

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

Figure 3.2: 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. SamplingFrequenciesTable class has been implemented to satisfy this need.

SamplingFrequenciesTable class

This structure is charged of representing the IF-ELSE-EVERY clause which allows to dynamically select sampling frequency. The internal state of this class is:

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

where FrequencyRecord is a 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. SamplingFrequenciesTable class exposes getter/setter methods to allow programmer obtaining every information about time based sampling. The last entry contained in a SamplingFrequencyTable class is relative to the ELSE branch and specifies the sampling behaviour that should be used when all the IF conditions are not satisfied. Finally, the class SamplingFrequencyTable also allows to specify the expected behaviour of the query when an unsupported sampling rate is required. PerLa currently foresees [1] two behaviours: SLOW DOWN and DO NOT SAMPLE. Such information can be retrieved using the getBehaviour() method.

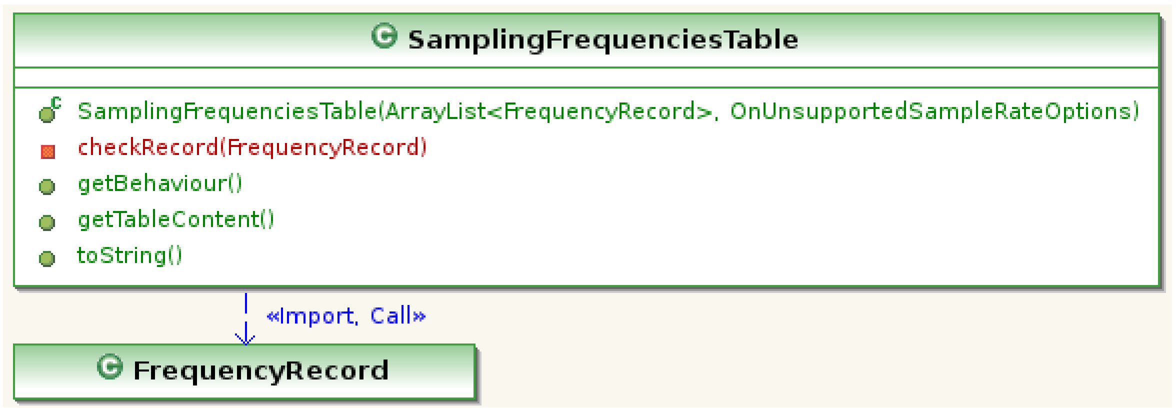

SamplingFrequenciesTable
SamplingFrequenciesTable(ArrayList<FrequencyRecord>, OnUnsupportedSampleRateOptions)
checkRecord(FrequencyRecord)
getBehaviour()
getTableContent()
toString()
«Import, Call»
FrequencyRecord

Figure 3.3: Data structure for sampling frequency UML diagram

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

LLQInformation class

This class wraps the "general" information of a single LLQ, spacing from data regarding the SELECT clause to data relative to the SAMPLING clause. More specifically, the class exposes the following information about a LLQ query:

- the expression representing the WHERE clause
- the references to a SamplingFrequenciesTable object (if time based sampling is adopted)

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

LLQEnvironment class

This class is deputed to represent the LLQ Environment component presented in Subsection 1.1.2.2. According to Figure 1.3 this class also contains references to:

- the LLQInformation class explained before
- a set of DataCollectors currently running and retrieving data from the pervasive system
- a set of local buffers, i.e. Buffer class (one for each DataCollector, as mentioned in 2.3)

- a reference to a future *LLQ Executor* which physically will compute LLQ results (this component still has to be designed and implemented, see also Section 2.2)

This class actually implements more than what requested in Section 2.4, where only one DataCollector and one local buffer were supposed. In fact, the sets of DataCollectors and local buffer reduce to contain a single element, implying a waste of resources using sets to contain only one single value. This choice was actually taken to enable the current implementation to be still functioning when hypothesis exposed in Section 2.3 will be removed (see also Chapter 4). Figure 3.4 represents both LLQEnvironment and LLQInformation classes as well as the implemented setter/getter methods.

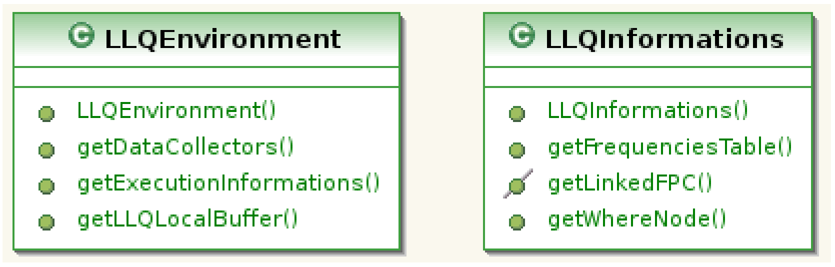


Figure 3.4: LLQEnvironment and LLQInformation UML diagram

3.1.3 Data structure for query analysis

As mentioned in Subsection 1.1.2.2 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 tractation something more must be said about *Query Analyzer*. Actually its activity is not only limited to DataCollector creation: LLQInformation class must be built and, together with a proper DataCollector, they must be inserted in a LLQEnvironment class. Exposing a complete set of methods to analyze queries, QueryAnalyzer summarily endorses the role of a *factory*: in fact, considering the whole set of exposed methods, programmers can obtain every type of information from a selected query and use it for their own purposes. This factory fashion is achieved defining the class methods as *static* (enabling them to be called without instantiating a QueryAnalyzer object). In the following a set of QueryAnalyzer methods is presented; note that Query is always the Java object representing the root of a query (as defined in [5]), while int parameter is the reference number of the LLQ: in fact, in PerLa more than one LLQ can be found in each query (see also Figure 1.1). As exposed in Figure 3.5, QueryAnalyzer class exposes the public method createLLQEnvironments (Query parQuery) which is an example of the *factory* behaviour described before: in fact, using a whole set of private methods (such as getWhereNode(), getSamplingTable() and so on), this method is able to:

1. create a set of DataCollectors, containing instances of the DataCollector object, one for each FPC involved in the query computation;

2. allocate a proper Buffer class, one for each DataCollector (as mentioned in Section 2.3);

3. instantiate LLQInformation class with all the information needed;

4. return to the user a Java ArrayList of LLQEnvironment classes instantiated using the information obtained at previous points, one for each LLQ detected by parser.

For a better comprehension both UML class and sequence diagram are reported respectively in Figure 3.5 and Figure 3.6.

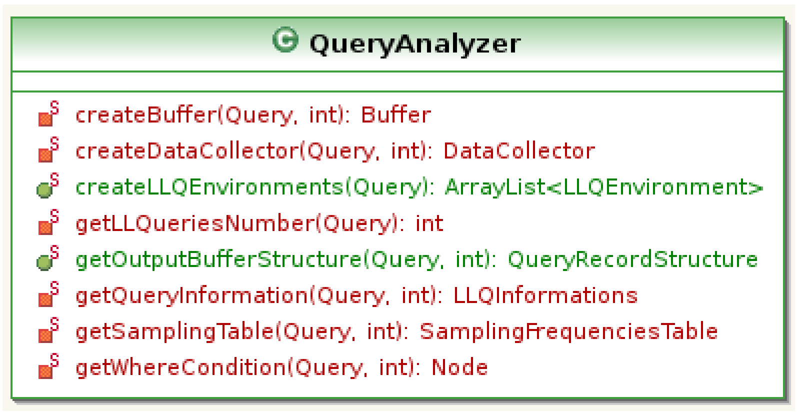

QueryAnalyzer
createBuffer(Query, int): Buffer
createDataCollector(Query, int): DataCollector
createLLQEnvironments(Query): ArrayList<LLQEnvironment>
getLLQueriesNumber(Query): int
getOutputBufferStructure(Query, int): QueryRecordStructure
getQueryInformation(Query, int): LLQInformations
getSamplingTable(Query, int): SamplingFrequenciesTable
getWhereCondition(Query, int): Node

Figure 3.5: QueryAnalyzer UML class diagram

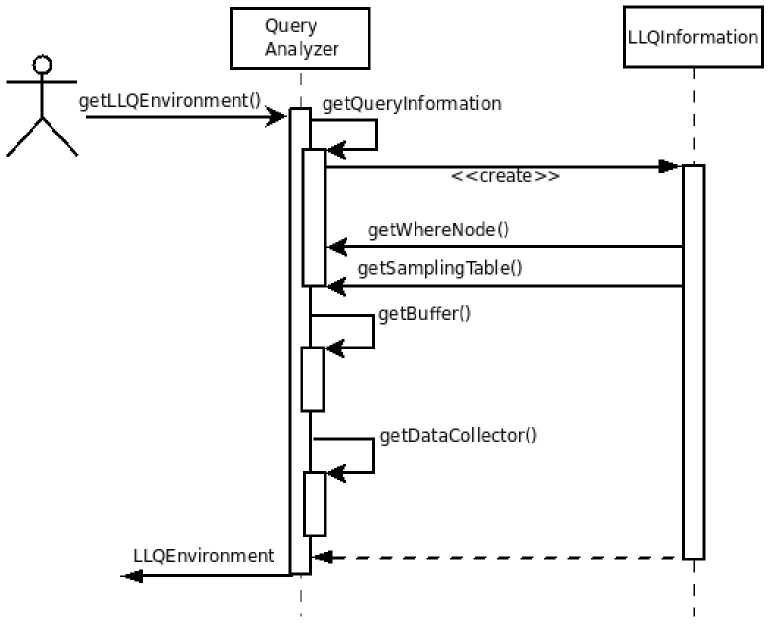


Figure 3.6: QueryAnalyzer UML sequence diagram

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 [9] 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);
```

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,

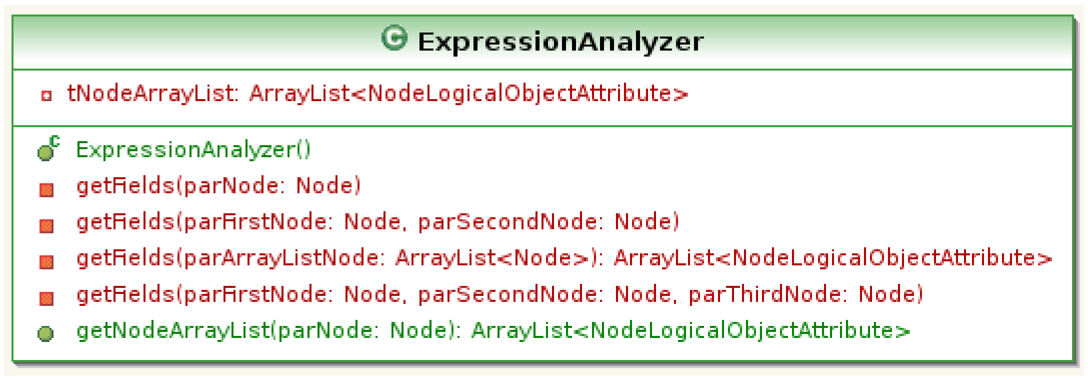

ExpressionAnalyzer
tNodeArrayList: ArrayList<NodeLogicalObjectAttribute>
ExpressionAnalyzer()
getFields(parNode: Node)
getFields(parFirstNode: Node, parSecondNode: Node)
getFields(parArrayListNode: ArrayList<Node>): ArrayList<NodeLogicalObjectAttribute>
getFields(parFirstNode: Node, parSecondNode: Node,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

Figure 3.7: ExpressionAnalyzer UML diagram

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 3.1.1): at this point attributes have been substituted and expression computation can be executed calling getResult() method of Node class, as implemented in [9]. The (boolean) result of the evaluation is then returned to the caller.

3.2 Data collection implementation

Before focusing on DataCollector implementation details, some preliminary considerations must be explained. When this project started, FPC implementation [4] was not completed yet: it can be useful to remember that FPC has to manage heterogeneity of the pervasive system making devices - middleware communication possible, and handling all the issues that such goal presents. To make DataCollector implementation possible the following framework (exposed in Figure 3.8) was created:

- **FakeFPC**: it's a mockup of an FPC including all the functions currently implemented. It will be substituted with the real implementation when project [4] is completed.
- **FakeDevice**: it's a Java class which represents a mockup of a sensor, and is charged to produce a record (composed of three fields) with a certain frequency. A method, called setSamplingFrequency(), is provided in order to allow the FakeDevice user to set the needed sampling frequency.

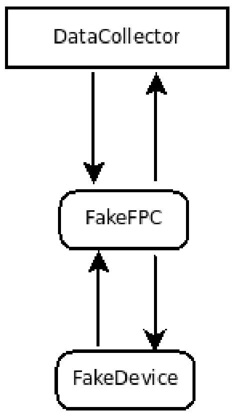


Figure 3.8: Implementation of simplified design

3.2.1 Communication structure implementation

Let's now focus on Java Classes that allow data exchange between FPC and DataCollector objects: these classes are called *pipes*.

Pipe class

Pipe class plays the role of a **synchronized** channel between two elements. Referring to Figure 3.9 let's explain how synchronization is achieved, supposing that component A has to send a message for component B.

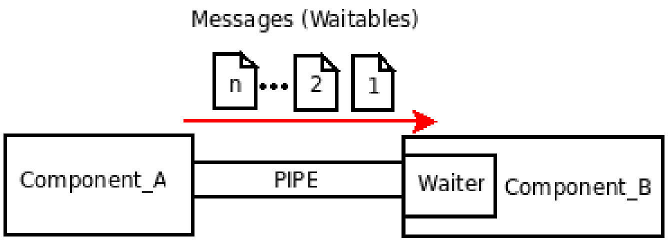


Figure 3.9: Pipes

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 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.

In this circumstance pipes are used to link all the components introduced at the beginning of Section 3.2, except for the method setSamplingFrequency(). An updated "version" of Figure 3.8 is shown in Figure 3.10: pipes are represented as blue arrows, while standard function calls are represented as black arrows. Pipes class are in fact meant to be used only when it's really necessary: setSamplingFrequency() method in fact does not require FakeFPC and FakeDevice to be decoupled nor require synchronization since it simply sets the value representing sampling frequency.

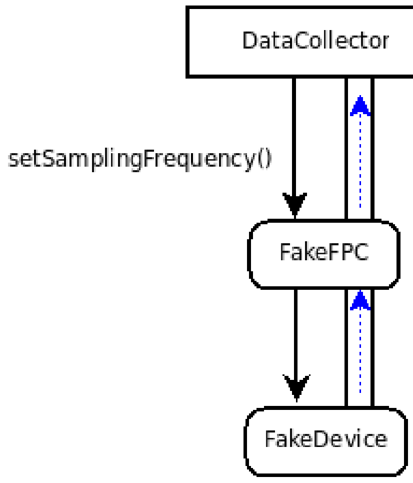


Figure 3.10: Simplified design schema using pipes

3.2.2 DataCollector class

DataCollector class implements the data collection activity proposed in Subsection 2.1.1, under the hypothesis exposed in Section 2.4. DataCollector lifecycle as Java object begins with its constructor, which has the following parameters:

- FPC parFPC: is the reference to the FPC bound to the DataCollector;
- Buffer parBuffer: is the reference to the local buffer where sampled data will be stored;
- Node parWhereNode: is the reference to the node representing the WHERE clause;
- SamplingFrequenciesTable parTable: is the reference to the data structure representing possible sampling frequencies;
- Pipe<Record> parInPipe: is the reference to the Pipe class that links DataCollector to the relative FPC.

DataCollector 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 from the Pipe that constructor received as parameter. 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 sets a new sampling frequency (calculated using parTable) on FakeDevice invoking setSamplingFrequency() method on FakeFPC. According to Figure 2.1, 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 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.

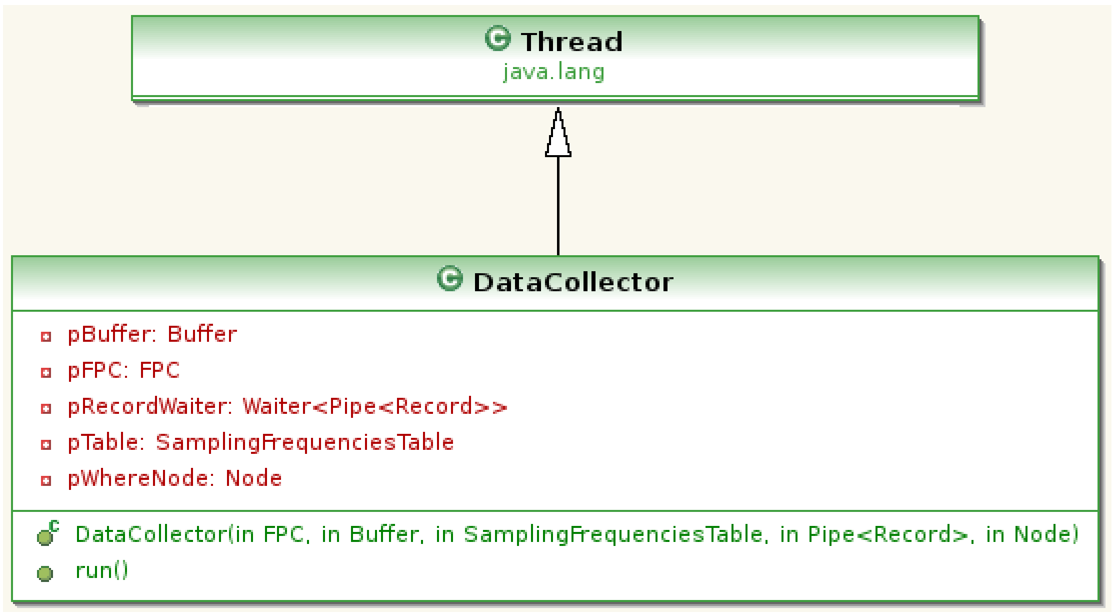

Thread
java.lang
DataCollector
pBuffer: Buffer
pFPC: FPC
pRecordWaiter: Waiter<Pipe<Record>>
pTable: SamplingFrequenciesTable
pWhereNode: Node
DataCollector(in FPC, in Buffer, in SamplingFrequenciesTable, in Pipe<Record>, in Node)
run()

Figure 3.11: DataCollector UML class diagram

4 Open points and future developments

As mentioned in the previous chapters, the management of collected data is still to be completed with the removal of the implementative hypothesis exposed in Section 2.4; moreover, interactions between data collection and data management parts of LLQ execution still requires design in order to give answers the following questions:

- Should there be a synchronization between the two parts? Certainly for sure, but how can it be achieved?
- How LLQ termination should be handled? How DataCollector component can signal the termination of a running query to the FPC? And how device errors should be handled in relation to the query termination?
- As mentioned in Chapter 2, in a real scenario more than one query can be executed simultaneously on the same FPC. In this work we supposed the existence of only one running query: the interactions among multiple DataCollector objects working on the same FPC must be investigated.
- Supposing that LLQ has been successfully executed how can results be pushed back to compute HLQ?
- QueryAnalyzer still requires the implementation of a certain number of methods, useful to obtain information about query computation (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 project scopes.
- 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.
- *Logical Object Registry* still requires to be completely designed and implemented, as briefly mentioned in this report.

Future developments should provide answers to the previous questions. Moreover, the FakeFPC component should be substituted with a real FPC implementation and the FakeDevice should be replaced by real sensors. 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 [12].

Appendix A: ExpressionEvaluator test

ExpressionEvaluator is a key element in DataCollector, since it allows the evaluation of expressions using a single record received, i.e. the FrequencyRecord objects inserted in the SamplingFrequenciesTable object. Physical computation is endorsed (and thus implemented) within Node and Constants implementation of [9]. A secondary goal of this project was to create a test framework in order to verify the correct implementation of these functionalities. The attention was focused on the most critical operations like division by zero and operations in three values logic. Tests are performed using the following syntax:

Node ConstantClass1 ConstantClass2 Value1 Value2 Result Result_type Exception (*)

Node represents the Node class to be tested (it should be noticed that each node is actually an operator between two Constant), while *ConstantClass1, ConstantClass2, Value1* and *Value2* are, respectively the arguments (data types and physical values) of the Node operator (i.e. the operands on which the computation is performed on). If the computation is expected to be completed without errors, *Exception* field can be left blank, otherwise *Result* is the expected result of computation. Otherwise, the expected Java Exception class can be specified if an error is expected. *Result_type* allows the framework user to verify the correctness of the type of the result, without effectively computing it (it's immediate to understand that a sum between two integers will be an integer, without physically adding the operands). The test framework has been designed and implemented charging a TestExecutor class to receive a text file containing a certain number of test lines structured as (*): for each parsed line (LineParser class), TestExecutor builds an ad-hoc class (SingleTest class) using Java *reflection* technique to instantiate all the elements (Node, Constants) required by the test. After single tests execution, results are reported to the user both in success and failure cases.

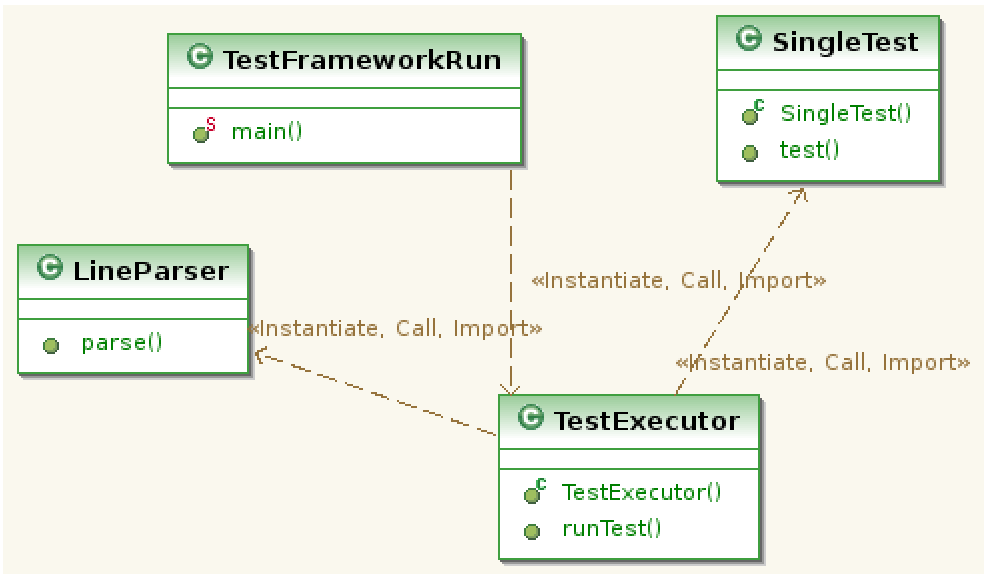


Figure 4.1: Test framework complete UML diagram

Bibliography

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

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

[3] 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.

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

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

[6] 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.

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

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

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

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

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

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

[13] TinyDB Website. http://www.tinyos.net, 2008.