# Chapter 18
# Extracting Data from WSNs: A Data-Oriented Approach

Fabio A. Schreiber, Romolo Camplani, and Guido Rota

**Abstract.** The PerLa language and the related middleware have been developed to ease the task of querying heterogeneous devices in pervasive systems. This paper presents, in a detailed way, some of the main features of the PerLa language by showing how it can be applied to the wine production process.

## 18.1 Introduction

The wine production process requires the cooperation of many different ”technologies” and expertises, which work in a pipelined manner: from the ”wine design”, performed by the oenologist, of the blending and timing for a quality wine, to the grape cultivation in the vineyard, up to the wine delivery to the consumer table, as discussed in chapter 17 and chapter 21.

As we can see in Figure 18.1, sensors and portable computing devices play an important role in each of the process steps; however the different environments and scopes require very different devices to gather and send data to the production control system, each with its own format and protocol.

Many different data are to be collected from sensors in the vineyard in order to control both the grape ripening conditions and the possible parasites attacks or on the barrels to control the wine ageing; even more differences among the information needed by the vineyard workers on their PDAs and on the RFID tags to be attached

Fabio A. Schreiber · Romolo Camplani
Politecnico di Milano, Dipartimento di Elettronica e Informazione, 34/5 Ponzio, 20133 Milano, Italy
e-mail: {schreiber,camplani}@elet.polimi.it

Guido Rota
Politecnico di Milano, Dipartimento di Elettronica e Informazione, 34/5 Ponzio, 20133 Milano, Italy
e-mail: guido.rota@gmail.com

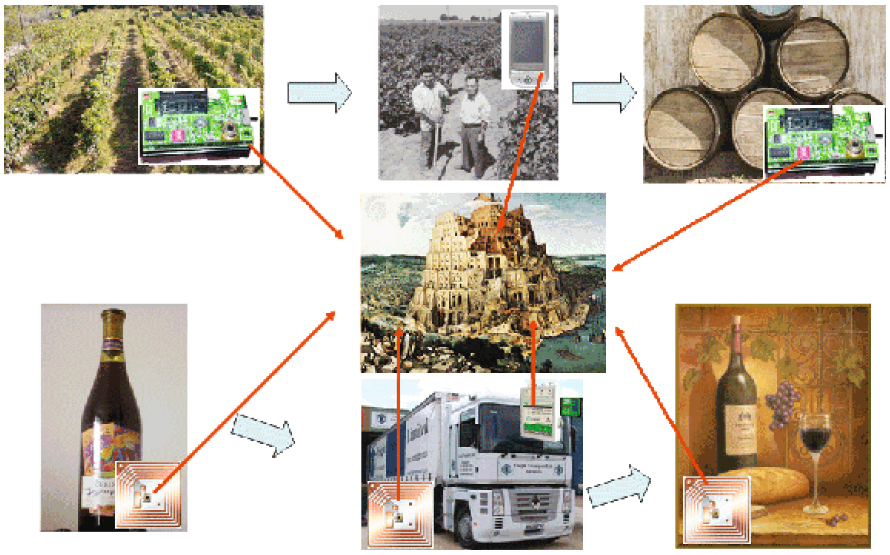


**Fig. 18.1** The wine production and delivery process

to the pallets or to each bottle. In this chapter we outline the winemaking monitoring process (see Figure 18.2), and we shall discuss how we overcome the challenges raised by the different sensing devices that support this process.

In this chapter we introduce *PerLa* - a language and middleware for data management in pervasive systems - by showing how its features can be applied to the winery scenario. In section 18.2 we introduce the monitoring requirements. In section 18.3, the main feature of the PerLa language are introduced by means of set of queries relevant to the wine production monitoring processes, while in section 18.3 a more formal presentation of the language is introduced.

## 18.2 The Vinification Monitoring Process

The task of controlling the vinification process begins in the vineyard. Humidity and temperature have a strong influence on the final quality of the wine, as they directly influence the grape's maturation process. Keeping these parameters under strict control is therefore an activity of paramount importance. The monitoring system, by means of sensors deployed in the vineyard, is able to provide continuous readings of humidity and temperature, as well as fire warning alarms should one of the parameters of interest exceed the ranges considered safe by the oenologists. Furthermore, to make full use of the resources available in the field, any relevant data sensed by the *PDAs* provided to the vinery workers is automatically integrated with the readings coming from the stationary sensor nodes.

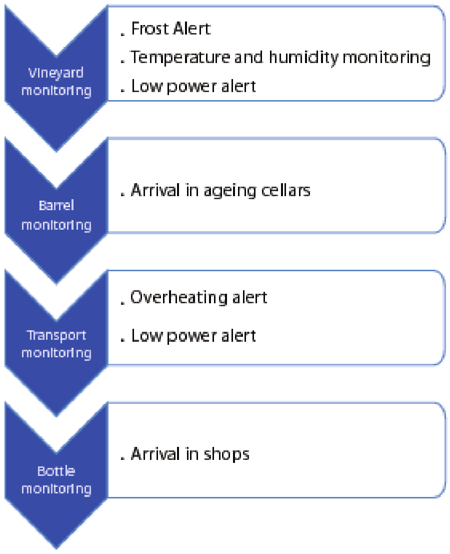


**Fig. 18.2** Wine production process monitoring - Workflow

The wine production monitoring process doesn't stop in the vineyard. Just after the grapes are harvested, and the fresh wine is made, the ageing process begins. The age and the peculiar character of the wooden barrels, the humidity of the ageing cellar, and unwanted temperature spikes are just few of the several factors involved in the development of wine flavour. The oenologists are required to constantly monitor all these aspects throughout the entire ageing period, to ensure the correct environmental conditions needed to mature the desired wine savour are maintained. The winemaking monitoring system is employed to chronicle the cellars where a wine barrel is stored during the ageing period. To this purpose, *RFID tags* are to be attached on every barrel, and the *RFID Readers* installed in the cellars will be used to trace the different locations where the wine is stored. This information can then be crossed with the data sensed by the cellar's legacy environmental control systems to obtain a complete monitoring of the wine maturation process.

To provide a detailed account of the entire production activity, it is essential to monitor the transportation phase as well. Since the flavour of wine can be easily spoilt by a sudden change of temperature, every pallet of bottles is to be provided with a thermal sensor during shipment. The wine will then be marked accordingly in the eventuality of an overheating. Moreover, by means of *GPS receivers* installed on the trucks, the monitoring system can assess if the current situation requires additional surveillance (e.g. when the payload is stationary in a sunny area), and ensures that the temperature is sampled with adequate frequency.

All the information gathered during the vinification is stored in a database for future evaluation. The complete account of the production process of a bottle of wine, made available to winemakers and the final consumers as well, can be retrieved by means of the identification code stored in the *RFID tag* located under the label of each bottle.

In the remainder of this chapter we will briefly outline the architecture of the aforementioned database and provide a short description of *PerLa* language and middleware, the system employed to collect data from the sensing devices. The decision to adopt *PerLa* is the results of a thorough analysis of current state of the art technologies for *Pervasive Systems* and *Wireless Sensor Networks* [1]. TinyDB [2], DNS [3], Cougar [4], Maté [5], Impala [6], Sina [7], DsWare [8], MaD-WiSe [9], Kairos [10], GSN [11], SWORD [12] and *PerLa* were evaluated to determine which one met the requirements of the winemaking monitoring system. *PerLa* has been chosen by virtue of its SQL-like declarative query language and *Plug & Play* node addition system, which greatly simplified the interactions with the sensing network devices.

## 18.3 PerLa: System Description

While in chapter 17 a service oriented approach is presented, *PerLa* is a language and a data processing middleware for *Pervasive Systems*, developed to mask the idiosyncrasies of the nodes employed in complex sensing networks based on the Database approach. The pervasive systems, exposed as a Database by *PerLa*, can be queried through a declarative language with SQL-like syntax. This feature allows application developers to release themselves from the burden of managing the peculiar behaviour of different sensing devices. Collecting data from a pervasive system abstracted by *PerLa* can be as easy as writing a typical database query. Moreover, the *PerLa* language [13][14][16] is composed of special syntactic statements designed to fully exploit the capabilities typical of pervasive sensing networks. Various examples of these statements will follow in the remainder of this chapter.

All nodes present in the sensing network are abstracted by the *PerLa middleware* as proxies called *FPC* (Functionality Proxy Component). These components have common and homogeneous interfaces, and are used by *PerLa* queries to access the data gathered from the network nodes. No knowledge of the node's hardware and computational characteristics is needed to perform a *PerLa* query. Moreover, by means of the FPC abstraction, the language is not tied to any particular type of sensing device.

*PerLa* support for pervasive systems extends to node developers as well. The addition of new sensing devices in an existing network is facilitated by a *Plug & Play* connection system, i.e. a runtime factory that generates all the software components needed to query new sensor nodes. The information required to automatically assemble a device driver are stored in an XML file drafted by the node developer. This file, dubbed *Device Descriptor*, details all the node's characteristics in terms of data structures, protocols of communication, computational capabilities, and behavioural patterns. The descriptor can be sent by the device itself upon startup or directly injected by the user (e.g. for RFIDs or other dumb devices). All running queries will automatically make use of every new node added in the system.

Results generated by *PerLa* queries are automatically stored in a relational database. This features allows third party software (see chapter 13) to easily access the data collected from the sensing network (see chapter 15 and 16).

*PerLa* middleware is being continuously updated. At the present time, a series of ongoing projects is aiming at expanding the system by adding an intelligent power management, new context-aware query statements and context-management features [15][17] and a virtualization layer to fully exploit the computational capabilities of the network nodes.

## *18.3.1 PerLa: Integration in the Winemaking Monitoring Process*

As the reader may have already realized from the introductory sections of this book, the wine production monitoring system makes use of data coming from a wide variety of heterogeneous sensing devices. *RFID* readers and tags, *PDAs*, legacy environmental control systems and ad-hoc sensors are just a few examples of the different nodes that support the winemaking control system. Developing a custom-made application merely to administer and query scores of heterogeneous sensing devices would not be an effective solution, since even the slightest change in the data acquisition network would surely entail a substantial rewrite of the system. Moreover, the resulting product could not be reused in any other domain of application.

All these considerations led to adopt *PerLa* as a middleware to decouple the core monitoring application from the Pervasive System used as a data source. The information retrieved from the sensing network is stored in the relational database described in chapter 9, and then analysed and processed by the wine production control system. A simplified version of this database schema is shown in Figure 18.3. While a detailed description of the PerLa language and its EBNF formal description can be found in [14], in the remainder of this section we shall describe the *PerLa* queries used for the monitoring process, and we shall use them to introduce the semantics of the *PerLa* language and the major software components of the *PerLa* middleware.

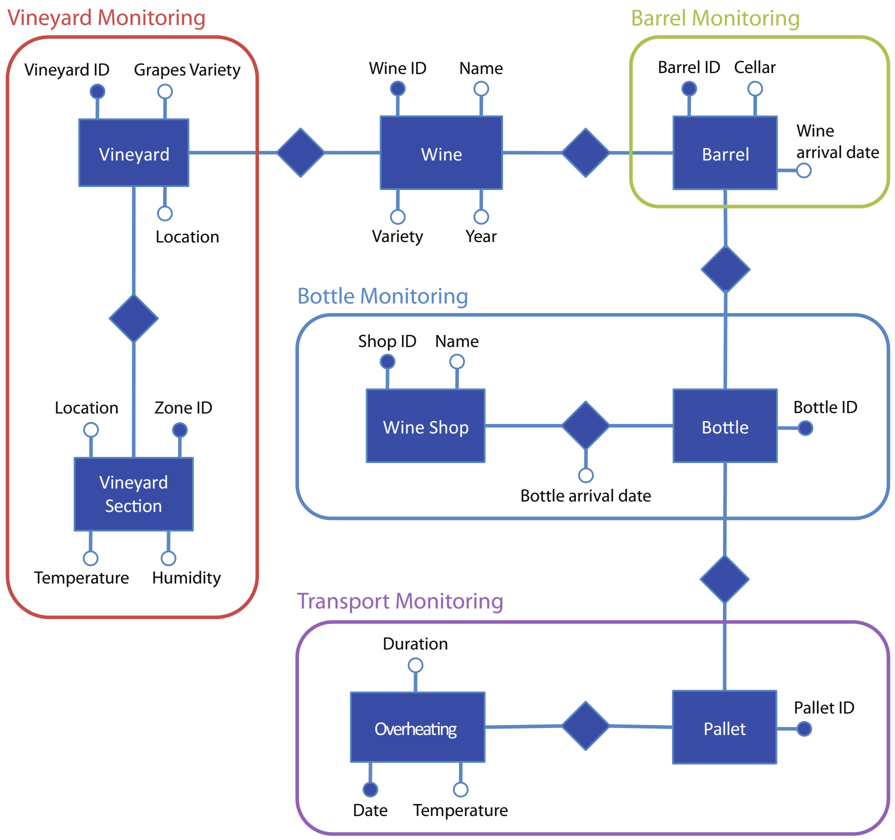


**Fig. 18.3** Wine production process monitoring - Database

```
1  CREATE OUTPUT STREAM Monitoring
2          (nodeId ID, temperature FLOAT, humidity FLOAT, locationX FLOAT, locationY FLOAT) AS
3  LOW:
4          EVERY ONE
5          SELECT ID, temperature, humidity, locationX, locationY
6          SAMPLING
7              EVERY 1 m
8          EXECUTE IF EXISTS (temperature) AND is_in_Vineyard (locationX, locationY)
9              REFRESH EVERY 10 m
```

**Fig. 18.4** Vineyard monitoring query

#### 18.3.1.1 Query A: Vineyard Monitoring

The purpose of this query is simple: collect temperature and humidity from all the nodes located in the vineyard.

The first 2 lines of Figure 18.4 contain the declaration of an *OUTPUT STREAM*, one of the two data structures available in *PerLa*. A *STREAM* is fundamentally an unbounded table, designed to be used mostly as an output data structure. As can be seen in the second line of Figure 18.4, every record of a *STREAM* is composed of a fixed set of fields, each of which has an identifier (nodeID, temperature, . . . ) and a type (ID, FLOAT, . . . ). In addition to the fields declared by the user, every record is provided with a native *TIMESTAMP* field.

The keyword *AS* (Figure 18.4, second line) is then used as a shorthand to indicate that the results of the query have to be stored in the previously declared data structure.

At the third row, the body of the query begins. *PerLa* supports two different types of queries:

- **Low Level Queries:** define the behaviour of a single sensing or actuation device. *Low Level Queries* allow the user to:
  - Set the sampling mode of the nodes
  - Execute SQL operations on the data sampled from the *Pervasive System*, (e.g. filtering, aggregations, . . . ).
- **High Level Queries:** perform SQL operations on *STREAMS* generated by *Low Level Queries* or other *High Level Queries*. Since the data extracted from the sensor nodes is stored in a relational database, the winemaking monitoring application does not use this type of queries extensively. An example of a *High Level query* is to be given in Fig.18.8.

*Low Level Queries* and *High Level Queries* are respectively identified by the keywords *LOW* and *HIGH*.

The *EVERY* clause (fourth line) specifies the execution condition of the Low Level *SELECT* statement. In this case, an event-based approach is chosen using the keyword *ONE*. As a result, the selection is scheduled to run every time a sample is gathered from the device. The *EVERY* clause, in addition to the event-based behaviour (*EVERY ONE*, *EVERY 2 SAMPLES*, . . . ), supports a time-based semantics as well (*EVERY 20 m*, *EVERY 2 h*, . . . ).

The selection clause (Figure 18.4, line 5) is then introduced by the keyword *SELECT*. The semantics of this clause is identical to its SQL counterpart, with just one difference: the data source of *PerLa* queries is not a database table, but a group of sensing devices instead. Depending on the peculiar characteristics of the nodes that compose the sensing network, same types of information may be collected by means of different techniques. The location of the devices employed in the vineyard monitoring, for example, can either be read from memory (if the node is known to be stationary) or sampled from a *GPS* receiver (if the device is a *PDA* assigned to a field worker). Despite these differences, the same selection statement (*SELECT locationX FLOAT, locationY FLOAT*) correctly retrieves the location from both devices.

The *SAMPLING* keyword (Figure 18.4, line 6) defines which sampling mode is required to collect the data requested in the *SELECT* clause. Two different semantics are available:

- **Time based:** the sampling frequency is set explicitly by the user. This mode is chosen by means of the keyword *EVERY*, followed by the desired time interval (e.g. 1 m, 10 m, 3 h, . . . ).

- **Event based:** the sampling operation is performed upon the occurrence of an event. The event based semantics, further discussed in one of the following queries, is activated with the keyword *ON EVENT* followed by the list of events chosen to trigger the sampling.

In the instance of Figure 18.4, this clause simply specifies a sampling interval of 1 minute.

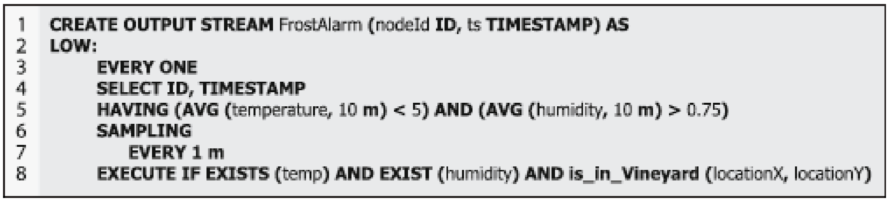
*PerLa* queries may also be used to describe other aspects of the data gathering process, empowering the user with the ability to greatly influence the operational behaviour of the single devices. It is worth mentioning that no knowledge about the hardware and software features of the sensing nodes is required when writing *PerLa* queries, even when dealing with clauses that govern the lowest layers of a pervasive system. We will delve further into this topic while describing the forthcoming queries.

The *EXECUTE IF* clause (Figure 18.4, line 8) is used to determine, by means of a logical expression, the set of devices on which the low level query is to be deployed. The statement of Figure 18.4 is set to be executed on all nodes located in the vineyard equipped with at least a temperature sensor. The vigilant reader may have noted that the selection clause demands a humidity reading, whereas the *EXECUTE IF* clause doesn't require the presence of the corresponding sensor. Whenever this query is executed on a device lacking the humidity transducer, a *NULL* value will be returned instead of the missing reading.

The *REFRESH* clause (line 9) forces the *EXECUTE IF* condition to be reassessed every 10 minutes, in order to update the list of devices capable of running the query. This allows the *PDAs* employed by the winemakers to be included when they access the vineyard. Should the *REFRESH* clause be omitted, the *EXECUTE IF* condition would be evaluated only once.

### 18.3.1.2 Query B: Frost Alarm

The second query, shown in Figure 18.5, is designed to monitor the environmental conditions that may induce frost in the vineyard. This phenomenon is known to show itself when temperature is near 0°C and the amount of water vapour in the air is significantly high. In this implementation of the query, the frost alarm is raised by adding a record in the *OUTPUT STREAM* whenever the temperature decreases below 5°C and the humidity in the air is over 75%.



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

**Fig. 18.5** Frost alarm

This particular behaviour is enforced by means of the selection statement of a *Low Level Query* (Figure 18.5, lines 4 and 5). The *HAVING* clause, which, in contrast with standard SQL language, works both for aggregates and single values as well, filters the data sampled by the network nodes and discards all the inappropriate records. The predicate used by this selection clause imposes a condition on both temperature and humidity readings, which evaluates true only when the vineyard is at risk of freezing.

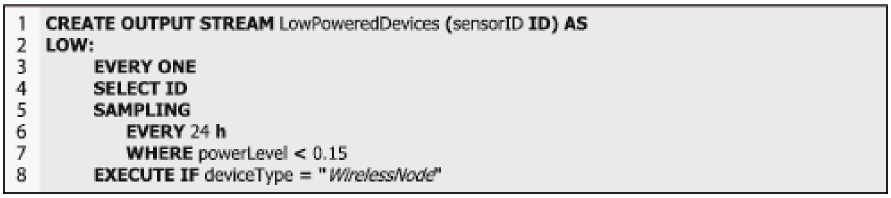
Note that the frost condition is evaluated on the average of the last 10 minutes samples in order to remove incidental noise. This is accomplished by using *PerLa* aggregate operators. Differently from their SQL counterpart, these operators have two parameters: the value on which the operation has to be carried out and the number of samples that are to be used to compute the aggregate. This difference from standard SQL aggregates is due to the peculiar nature of the input data sources. The information collected from pervasive system nodes is a continuous stream of data. Specifying to which extent the aggregation operation should be performed is therefore mandatory, otherwise the computation would never terminate due to the potentially infinite data set. The number of samples on which the aggregate is calculated can be specified either as an explicit number (e.g. *AVG(temperature, 25 SAMPLES)*) or as a time duration (e.g. *AVG(temperature, 10 m)*).

Since air humidity is fundamental to forecast the frost phenomenon, the *EXECUTE IF* clause (Figure 18.5, line 8) mandates the presence of the corresponding transducer on all the nodes involved in the execution of this query. By contrast, this sensor was tagged as optional in the first query (Figure 18.4, line 8).

#### 18.3.1.3 Query C: Devices with Low Battery

The state of a sensing network managed by *PerLa* is abstracted as a set of records collected from a group of potentially virtual sensors. Therefore, non functional information regarding the network nodes such as battery status, processor speed, software or hardware revision, etc. can be accessed by means of a plain *PerLa* query. No special statements or clauses are needed for this purpose.

The instance of Figure 18.6 exploit this feature to list the identifiers (*IDs*) that belong to wireless devices with low residual battery charge.



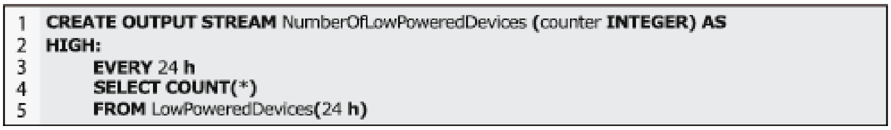
```
CREATE OUTPUT STREAM LowPoweredDevices (sensorID ID) AS
LOW:
      EVERY ONE
      SELECT ID
      SAMPLING
          EVERY 24 h
          WHERE powerLevel < 0.15
      EXECUTE IF deviceType = "WirelessNode"
```

**Fig. 18.6** Low Powered Devices

In contrast with the examples shown up to this point, the query of Figure 18.6 employs the *SAMPLING* clause to determine whether a node's battery is nearing exhaustion (line 7, *WHERE powerLevel < 0.15*). Previous examples (e.g. Figure 18.5) relied entirely on the *SELECT ... HAVING* syntax to filter records; the usage of the *SAMPLING ... WHERE* construct produces the following effects:

- the discarded records are not processed by the *SELECT* statement
- the discarded records do not trigger the execution of the *SELECT* statement when the execution condition is event based (e.g. *EVERY ONE, EVERY 2 SAMPLES, ...*)
- the records that does not fulfill the *WHERE* criterion are discarded by the sensor node itself; no data is transmitted over the network.

It is worth mentioning that the *SAMPLING ... WHERE* and *SELECT ... HAVING* constructs are not interchangeable, even though most trivial *PerLa* queries can be written using either of them. The *HAVING* clause is meant to be used when the filtering condition involves aggregate operations or other functions that require two or more records to be computed. The *WHERE* clause is a better choice if the filtering operation can be performed evaluating a single record. The latter approach can lead to a significant performance improvement, since the discarded values are not processed by the *SELECT* statement. Therefore, the *WHERE* clause should be preferred over the *HAVING* one whenever possible.



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

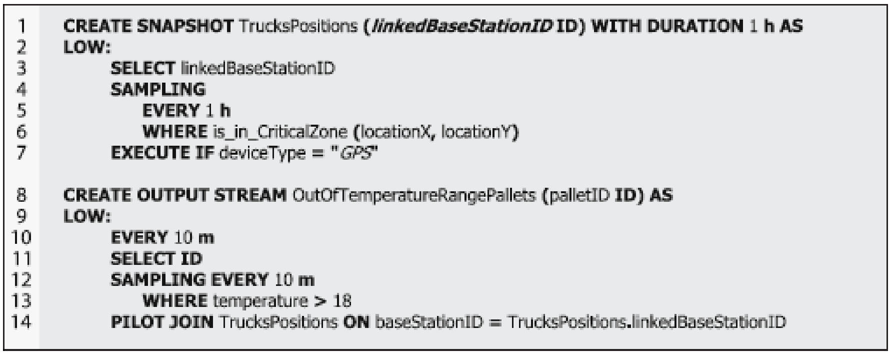
**Fig. 18.7** Number of low powered devices

The number of devices that need a battery replacement may be computed with the query of Figure 18.7. The statement, whose syntax closely mirrors standard SQL, is a simple *High Level Query*. Like the foregoing *Low Level Queries*, the selection statement activation condition is expressed via the *EVERY* clause. In this instance, the query is run every 24 hours (Figure 18.7, line 3). The only operation required to reckon the number of devices in need of a new battery is a simple *COUNT(*)*. The raw data regarding the battery conditions of the network nodes are retrieved from the *LowPoweredDevice STREAM*, which is constantly updated by the query shown in Figure 18.6.

One of the major differences among SQL queries and *PerLa High Level Queries* lies in the *FROM* clause. As can be seen at line 5 of Figure 18.7, the input *STREAM* name is complemented with a duration (namely *Window Size*). This information determines how many records are to be processed whenever the *SELECT* statement is activated. The *Window Size* can be specified either as a time interval or a number of records.

#### 18.3.1.4 Query D: Pallets Out of Temperature

High temperatures can yield devastating effects on wine flavour. Even if protracted for a short amount of time, the exposure to a warm environment may irreversibly change the product's typical character and savour. These alterations are to be prevented at all costs to avoid wine depreciation and to increase customer satisfaction. To this purpose, the query in Figure 18.8 is employed to signal every thermal shock experienced by the wine during shipment. Considering that the thermal sensors employed in this application are powered by an autonomous battery, this query has been designed to minimize unnecessary data communications.



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

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

**Fig. 18.8** Pallets out of temperature

The shipment monitoring system is composed of these main elements:

- A temperature sensor node with low-range radio transmitter installed on every pallet of wine
- A *GPS* receiver installed on every truck of the shipment fleet
- A *Base Station*, i.e. a special network node used to relay the data from *GPS* and temperature sensors to the monitoring headquarters. Every truck is provided with a single base station.
- The *PerLa* query of Figure 18.8

Since under normal circumstances the trailer's air cooling system is considered adequate to safeguard the integrity of the wine, the continuous temperature monitoring is activated only when the shipment travels across a critical location. This behaviour is achieved through the *PILOT JOIN* execution condition.

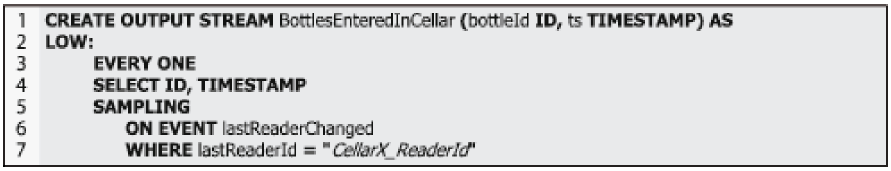
With *PILOT JOIN*, the decision to include a node in the execution of a query is based on the content of a support data structure. Therefore, the output of any sensing device can be employed to trigger the activation of a query on other network nodes. By contrast, the *EXECUTE IF* only allows the use of attributes gathered from a single device to determine whether that node could take part in a query or not.

The example in Figure 18.8 makes use of the *PILOT JOIN* clause to activate a monitoring query on those pallets that are traversing a critical location. To obtain this behaviour, a first subquery (lines 1 to 7) is run to gather the *IDs* of the *Base Stations* installed on trucks considered at risk of overheating. Then, the second subquery (lines 8 to 14) is used to collect the identifier of the pallets whose temperature exceeded the safe threshold (line 13). The *PILOT JOIN* (Figure 18.8, line 14) ensures that the scope of the monitoring query is limited to the bottles inside a critical location.

The first subquery, in contrast with all the preceding examples, is used to insert records in a *SNAPSHOT* table (Figure 18.8, first line). This type of data structure is intended to hold data for a limited time only, known as *SNAPSHOT DURATION*. Upon expiration, the *SNAPSHOT* is purged and filled with new records. A *PILOT JOIN* execution condition, if used in combination with a *SNAPSHOT* table, is evaluated when the content of the data structure is refreshed (conditional semantics). Were the same *PILOT JOIN* used with a *STREAM*, the evaluation of the execution condition would be triggered by every new record added in the data structure (event based semantics).

#### 18.3.1.5 Query E: Cellar Monitoring

The wine monitoring process does not stop once the product is delivered. To ensure that flavour and aroma are preserved, the wine has to be stored in a controlled environment, where unexpected variations in temperature, humidity or light are kept under strict control. All wine cellars must therefore provide some sort of controlled environment, either by nature or by means of a climate control systems. To complete the chain of trust between the winemaker and the final buyer, the wine dealers must be able to provide the list of cellars where every bottle in their catalogue has been stored. The query of (Figure 18.9) is specifically designed for this purpose.



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

**Fig. 18.9** Cellar monitoring

Since every bottle is labelled with an *RFID*, the content of every wine cellar can be easily monitored by installing a tag reader at the entrance door. The instance in Figure 18.9 is then executed to maintain a log with the *IDs* of all the wine bottles that crossed the threshold of the storing room.

In this situation, a continuous monitoring of the *RFID reader* would not make sense. Hence the decision to use an event based sampling technique. As explained in the description of the first query, the event based sampling mode triggers a reading from a sensing device upon the occurrence of one or more events. In the query of Figure 18.9, the sampling operation is set to be performed whenever the *RFID tag* of a wine bottle is sensed by a new *RFID reader* (i.e. the bottle has been moved to a new cellar). The *WHERE* clause of line 7 is additionally specified to monitor one cellar at a time.

The information produced by this query, when crossed with the data collected by the cellar's legacy environmental control system, provides a comprehensive chronicle of the wine storage phase. The final consumer can exploit this information to determine the exact conditions endured by the wine bottle during its entire lifetime.

It is worth mentioning that this query can be easily adapted to monitor the wine barrels introduced in the ageing cellars.

## 18.4 PerLa Language Digest

We conclude this chapter with a brief digest on the *PerLa language*, intended to give the reader a broad vision over the most important language statements and features [14].

### *18.4.1 Data Definition*

*PerLa* language provides the user with two distinct table types:

- **STREAM:** a table composed of an unbounded number of records. *STREAMS* represent *PerLa's* main data structure.
- **SNAPSHOT:** a set of records generated during a specified time period (*SNAPSHOT DURATION*). When the given duration expires, the *SNAPSHOT* is cleared and filled with new records.

*PerLa Data Definition* statements are introduced by the keyword *CREATE*, followed by the identifier of the desired data structure. Both *STREAMs* and *SNAPSHOTs* are a homogeneous collection of records. The record structure declaration, defined as a list of identifiers and data types, is therefore mandatory. *PerLa* tables can be additionally tagged with the keyword *OUTPUT* if their content is to be shown to the user who submitted the query.

A *STREAM* or *SNAPSHOT* definition is usually followed by a *PerLa* query, introduced via the keyword *AS*, which is executed to generate the data structure content.

### *18.4.2 Low Level Queries*

*Low Level Queries* are used to access the data produced by the sensing network. By means of this type of statement, *PerLa* users can define which information is to be gathered from the pervasive system , set the sampling mode of the sensing devices, select the network nodes on which execute the query, filter data and perform simple *SQL* operations. *Low Level Queries* are identified by the keyword *LOW*.

### Sampling

The *SAMPLING* clause is used to specify the behaviour of every single sensing device. Two different semantics are available:

- **Time based:** the sampling operation is performed periodically, following user indications. The sampling period can be specified as a fixed duration (*SAMPLING EVERY 3 m*), as a numeric expression (*SAMPLING EVERY (1000/temperature) s)*) or as a conditional statement (*SAMPLING IF powerLevel > 50 EVERY 1 s ELSE EVERY 15 s REFRESH EVERY 1 h*).
- **Event based:** the sampling operation is triggered by the occurrence of one or more events (e.g. SAMPLING ON EVENT highTemperature, lowTemperature).

The *SAMPLING* clause may also be employed to perform basic filtering operations (SAMPLE EVERY 1 m WHERE temperature > 50).

### Data Managing

*PerLa Low Level Queries* are provided with a *Data Managing Section*, introduced by the *SELECT* keyword, through which users define the exact output of the query. Legit *SELECT* statements can make use of constants (SELECT 3, ”Temperature”), data sampled from sensors (SELECT temperature, humidity, powerLevel), numerical expressions (SELECT $0.55 * (farenheitTemp - 32)$) or aggregate operations (SELECT timestamp, AVG(temperature, 10 m)). Two different activation semantics are available:

- **Time based:** query results are computed periodically (EVERY 10 m SELECT . . . )
- **Event based:** query results are computed when a determined number of records is available (EVERY 5 SAMPLES SELECT . . . )

*SELECT* statements can be complemented with a *HAVING* clause to specify advanced filtering conditions.

### Execution Conditions

This optional section is used to define which sensing devices can partake in executing a query. Execution conditions are expressed through the following clauses:

- **EXECUTE IF:** allows the definition of a simple predicate, which is evaluated to determine whether a node can take part in a query or not. Execution conditions set via the *EXECUTE IF* clause are reassessed periodically if the *REFRESH* keyword is specified.
- **PILOT JOIN:** provides the user with the ability to declare sophisticated execution conditions based on a support query.

Termination Conditions

The optional *TERMINATE AFTER* clause may be employed to declare the lifetime of a *Low Level Query*. Queries can be set to stop at the end a specified time period (e.g. *TERMINATE AFTER 1 h*) or after the selection statement has been executed an established number of times (e.g. *TERMINATE AFTER 10 SELECTIONS*).

### *18.4.3 High Level Queries*

*High Level Queries* are designed to perform data manipulation operations over *STREAM* tables. Syntax and semantics of this type of statement is closely related to standard SQL. There are, however, two major differences:

- **Activation conditions:** the *High Level* selection statements can be activated periodically (*EVERY 10 m*) or by the insertion of a new record in a *STREAM* (*EVERY 3 SAMPLES IN <StreamName>*)
- **FROM clause:** when using *STREAM* tables, the user is required to define a duration window that identifies how many records are processed during selection

*High Level Queries* are identified by the keyword *HIGH*.

### *18.4.4 Actuation Queries*

*Actuation Queries* are employed to set parameters on network nodes. This type of statement is mainly used to drive mechanical and electronic actuators or to modify software variables. *Actuation Queries'* syntax is composed of the keyword *SET*, which introduces the parameter to set, and the keyword *ON*, used to list the nodes interested by the query.

## 18.5 Final Remarks

The PerLa middleware has been adopted in a prototypical deployment for vineyard monitoring. In particular, we deployed half a dozen nodes, each one composed by a ZigBee-compliant Jennic JN39R131 mote and endowed with humidity, temperature and luminosity sensors. For the tests, we fixed the sampling rate to 1Hz (which is considerably high for the chosen scenario).

Despite the considered testbed is a significant case study for our system, further improvements may be only obtained by studying the scalability of the entire systems. In particular, we want to focus on the scalability both in terms of number of nodes per network and of number of served networks.

**Acknowledgements.** We thankfully acknowledge the work of Ing. Marco Fortunato and Ing. Marco Marelli, who pioneered the design and development of *PerLa*, and the contribution of Ing. Diego Viganò, who assisted us in the development of the system.

## References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. IEEE Communications Magazine 40(8), 102–114 (2002)
2. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1), 122–173 (2005)
3. Chu, D., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I.: The design and implementation of a declarative sensor network system. In: SenSys 2007: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, pp. 175–188. ACM, New York (2007)
4. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. 31(3), 9–18 (2002)
5. Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. SIGPLAN Not. 37(10), 85–95 (2002)
6. Liu, T., Martonosi, M.: Impala: a middleware system for managing autonomic, parallel sensor systems. In: PPoPP 2003: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 107–118. ACM Press, New York (2003)
7. Srisathapornphat, C., Jaikaeo, C., Shen, C.-C.: Sensor information networking architecture. In: ICPP 2000: Proceedings of the 2000 International Workshop on Parallel Processing, p. 23. IEEE Computer Society, Washington, DC (2000)
8. Li, S., Son, S.H., Stankovic, J.A.: Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In: Zhao, F., Guibas, L.J. (eds.) IPSN 2003. LNCS, vol. 2634, pp. 502–517. Springer, Heidelberg (2003)
9. Amato, G., Baronti, P., Chessa, S.: Mad-wise: programming and accessing data in a wireless sensor network. In: Proceedings of the International Conference on Computer as a tool EUROCON 2005 (2005)
10. Gummadi, R., Kothari, N., Millstein, T., Govindan, R.: Kairos: A macroprogramming system for wireless sensor networks. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles SOSP 2005 (2005)
11. Aberer, K., Hauswirth, M., Salehi, A.: Global sensor networks. School of Computer and Communication Sciences Ecole Polytechnique Federale de Lausanne (EPFL), Tech. Rep. LSIR-REPORT-2006-001 (2006)
12. `http://webdoc.siemens.it/CP/SIS/Press/SWORD.htm`
13. Schreiber, F.A., Camplani, R., Fortunato, M., Marelli, M., Pacifici, F.: Perla: A data language for pervasive systems. In: Proc. PerCom, pp. 282–287 (2008)
14. Perla Home Page, `http://perlawsn.sourceforge.net/`
15. Bolchini, C., Curino, C.A., Orsi, G., Quintarelli, E., Rossato, R., Schreiber, F.A., Tanca, L.: And what can context do for data? Communications of ACM 52(11), 136–140 (2009)

16. Schreiber, F.A., Camplani, R., Fortunato, M., Marelli, M., Rota, G.: Perla: A language and middleware architecture for data management and integration in pervasive information systems. IEEE Transactions on Software Engineering, doi:10.1109/TSE.2011.25
17. Schreiber, F.A., Tanca, L., Camplani, R., Viganó, D.: Towards autonomic pervasive systems: the PerLa context language. In: Electronic Proceedings of the 6th International Workshop on Networking Meets Databases (Co-located with SIGMOD 2011), Athens, pp. 1–7 (2011), http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final4.pdf