INSTALLATION AND USER MANUAL

V. 0.1

March, 2013

A cura di

FRANCESCO MARIA FILIPAZZI

MATR 726517

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Piazza Leonardo da Vinci, 32 - Milano 20133 Italy

TEL No. (+39)(2) 2399 3400   FAX No. (+39) (2) 2399 3411

Acknowledgments

The design and development of PerLa has been the effort of many B.Eng., M.Eng, and PhD. Students who worked at it in fulfilment of their thesis work under the supervision of Romolo Camplani PhD.

I want in particular acknowledge the work of Marco Fortunato and Marco Marelli, who pioneered the design and implementation phase, Guido Rota, who gave an appreciated engineering contribution, and Diego Viganò, who developed the Context-aware extension of the language and of the middleware, to whom goes my warmest thanks.

I also want to acknowledge the projects which supported our work:

MIUR-FIRB - ArtDeco
Adaptive Infrastructures for Decentralised Organisations

Politecnico di Milano - PROMETEO
Public Protection: Methodologies and Technologies

Politecnico di Milano - SMScom

Self-Managing Situated Computing

Prof. Ing. Fabio A. Schreiber
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

PerLa Project Manager

TABLE OF CONTENT

Acknowledgments 3

GOAL OF THIS DOCUMENT 4

INTRODUCTION TO THE PERLA SYSTEM 5

The PerLa language 5

The PerLa middleware 6

HOW TO START 8

Downloading and installing the system 8

Installing Perla 9

DATA STRUCTURES IMPLEMENTATION 10

Query implementation 10

Data Structures Computation...12

FROM PHYSICAL DEVICE TO FPC 14

The components 14

System working mode 15

Device to FPC transmission structures 18

FROM QUERY TO LOW LEVEL QUERY 19

FROM FPC TO QUERY ANALYZER 21

REFERENCES...25

GOAL OF THIS DOCUMENT

Goal of this document is to introduce a user of PerLa to the operational functions of the system.

Paper [01] contains a description of the full system while a detailed description of the language, its formal definition, and usage examples can be found in [02] while in [03] a description of the Context-aware features extension is reported.

INTRODUCTION TO THE PERLA SYSTEM

The PerLa language

There are many real word applications that are continuously monitored using a large number of heterogeneous sensing devices. Different kinds of sensors, both in terms of technology and functionality, are often simultaneously used within the same application. The integration of data collected using different technologies (e.g.: Wireless Sensor Networks, RFID tag and GPS) is certainly an interesting challenge, but the different interfaces provided to control and query each involved kind of device make this goal very hard to achieve. The PerLa project aims at defining a declarative high level language that allows to query a pervasive system, hiding the difficulties related to the need of handling different technologies. We provide a database like abstraction of the whole network in order to hide the high complexity of low level programming and allowing users to retrieve data from the system in a fast and easy way.

Taking into account the large heterogeneity of the considered devices, the language is actually split into three sub-languages:

* A LOW LEVEL LANGUAGE, that allows to precisely define the behavior of a single device. The main role of a low level statement is to define the sampling operations, but also to allow the application of some SQL operators (grouping, aggregation, filtering) on sampled data.

* A HIGH LEVEL LANGUAGE, that allows to define data manipulation over the streams coming from low level or other high level queries.

* An ACTUATION LANGUAGE, that allows to send some commands to the device they are executed on. For example, suppose that a logical object is wrapping an "ad hoc" board that performs signal filtering to generate sampled data: in this case, an actuation query could be used to change the filter parameters.

A fourth CONTEXT MANAGEMENT sub-language is being designed to deal with context-aware systems in order to define, create and activate contexts and implement context-dependent behaviours.

All sub-languages have an SQL like syntax, but the low level language semantics is quite different with respect to the standard SQL one: in fact, specific clauses have been introduced to manage sampling operations. Moreover, the language has been designed to make easier the generation of aggregated data starting from sampled data. On the contrary, the high level language semantics is very similar to that of streaming databases. The definition of the language semantics is completely based on the concept of LOGICAL OBJECT. This is an abstraction of physical devices that allows application objects to interact with the hardware, through the definition of a suitable interface. Each logical object wraps a single or a homogeneous aggregate of physical devices. At the application layer, a query analyzer handles user submitted queries and retrieves the list of logical objects

composing the system, using a specific component (registry). This information is then used to select the logical objects that will take part in the query.

The described architecture allows to abstract the whole pervasive system as a collection of logical objects. In this way, each request submitted to a node can be thought as a request to the corresponding logical object, through the exposed interface. The precise definition of this interface allows to describe the language semantics as a set of interactions between the query executor and the logical objects. As an example, from the language point of view, a sampling request to a node having a temperature sensor on board is abstracted as an attribute reading from the node interface. Similarly, when an RFID tag is sensed by a certain RFID reader, the language detects an event fired by the logical object wrapping the tag. This also allows to treat functional and non-functional attributes in the same way.

The PerLa middleware

The PerLa system is basically composed of three components:

The LANGUAGE, the goal of which is to enable the final user to collect data in a fast and easy way, without dealing with low level programming issues.

The NODES which are heterogeneous devices equipped with sensors, able to collect data and to send them in the network managed by the middleware. Nodes can either be very simple devices (such as RFID tags or WSN nodes such as MOTES) or more complex devices (such as palms, portable computers or ad hoc boards).

The MIDDLEWARE, the goal of which is to provide an abstraction for each device in terms of logical objects and to support the execution of PerLa queries. The middleware also implements a set of functionalities to allow the communications among logical objects, to manage devices that enter and leave the system (following a "Plug and Play" behaviour), and to perform the context management operations.

The definition and the addition of new devices is made easier by minimizing the amount of low level code (a small XML file) the user has to write to make the new device recognizable by the system.

The middleware is composed of the following layers:

LANGUAGE PARSER. It receives textual queries as input, verifies their syntax and transforms them in a suitable format for distribution and execution.

LOGICAL OBJECT REGISTRY. It is the component that maintains the list of the logical objects currently registered in the system. It is also used to find the set of logical objects that will be involved in the execution of a certain low level query.

HIGH LEVEL QUERY EXECUTOR. Basically, it is a data streaming management systems (DSMS).

LOW LEVEL SOFTWARE. This software layer adapts the different physical devices to the middleware. It also provides the abstraction of logical objects. We decided to implement the whole software, from the logical objects layer up, using JAVA technology. We also assumed that each logical object (that we can now consider as a JAVA remote object) is reachable via TCP/IP. If the physical device is connected to a different network (e.g. a CAN-BUS channel), the correspondent logical object will be instantiated on the nearest device equipped with both a JAVA Virtual Machine and a TCP/IP connection to the network managed by the middleware. This middleware layer implements the communication protocol between the logical object and the physical device.

LOW LEVEL QUERY EXECUTOR. This is a JAVA component, contained in each logical object, whose goal is to receive and execute a parsed low level query.

HOW TO START

Downloading and installing the system

Prerequistes for Perla

-It runs on Linux, iOs and others Unix-like systems;

-It is written in Java Programming Language;

-It uses **Jaxb**, a Java Library used to process XSD schemas;

-It uses a **Mysql** Database;

-**JavaCC** must be installed on system;

-The recommended development environment is **Eclipse with Maven** plugin.

JVM and JDK: Perla Requires the official Java Virtual Machine (www.java.net) and Java Development Kit. Different implementation of those software, such as OpenJDK, are deprecated.

Jaxb: Jaxb means "Java Architecture for XML Binding"[06]. Perla requires the Jaxb architecture to generate the data structures which represent devices and sensors of Pervasive System.

To start Perla, the user or the developer must install Jaxb and insert it in the Java Virtual Machine installed on his system.

Jaxb can be downloaded at jaxb.java.net.

JavaCC: This is a parser generator. It is a tool that reads a grammar specification and converts it to a Java program that can recognizes matches to the grammar.

To get information on it go to javacc.java.net.

Example: to install JavaCC on Ubuntu write "sudo apt-get install javacc" in the shell.

Maven: Maven is a project manager developed by Apache. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository.

Eclipse with Maven

To integrate Maven in Eclipse, go to “Help->Install new Software”.

Then add the m2e repository: maven - http://download.eclipse.org/technology/m2e/releases and install the plugin.

To create a maven project, go to File->New Project->Maven->Maven Project.

Mysql: is a relational Database Management System

Java Reflection: Perla uses the Reflection Api. Before reading or writing code, the user is suggested to read this tutorial:

http://docs.oracle.com/javase/tutorial/reflect/

The most used method and classes of Reflection Api in Perla are:

Classes: Object, Class, Field.

Methods: getClass(), getFields(), getDeclaredField().

Installing Perla

To install the last version of Perla follow these steps:

Download the code from perlawsn.svn.sourceforge.net/viewvc/perlawsn/experimental/0.9.9/ and import it in a Maven project created with Eclipse.

The imported project could be signed with an exclamation point. To solve this problem control the build path (right click on project->java build path->libraries).

Execute the jaxbDoGeneration.sh in terminal and the WsnPdl.jj (org/dei/perla/parser/grammar) file with javacc library.

Create a database with MySql. You can use a graphic interface or the command line of MySql.

Then set the database data in conf/wsnshell.conf file.

In place of “My_Perla” set the name of your database.

In PERLA_DB_AUTH write the user name used to open the database and in DB_PASS set the password.

After these steps the project will be ready to work.

DATA STRUCTURES IMPLEMENTATION

Query implementation

Data Structures are used in Perla to represent data sampled by FPC.

The first structure explained is the Record structure.

Sampled values 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.

Class: Query Field Structure
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 value contained in the field (ConstantFloat.class, ConstantInteger.class ...), the second represents the variable name (temperature, height ...) and the third represents the number of fields contained in the record.
To call the QueryFieldStructure constructor, these three variables must be passed. In the class, getters and setters of each variable are contained.

Class: Query Record Structure
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.

In this class two important methods are also implemented.
The first is

```
public void addFieldStructure(String parFieldName, Class<?> parFieldType)
```

that, given a QueryFieldStructure, adds a field to the current record.

The second is

public void mapThatPojo(Class pojo)

that takes an object or a class and converts it in a constant type. MapThatPojo() passes the converted data to addFieldStructure().

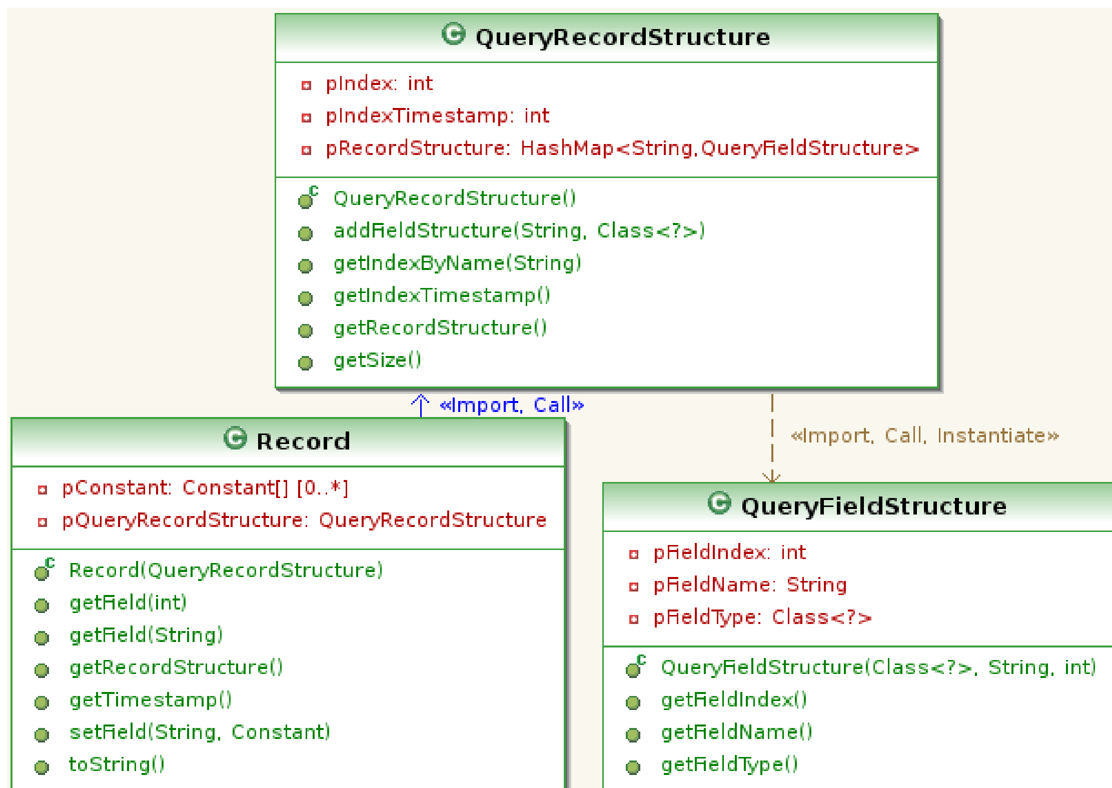
Class: Record

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

private Constant[] pConstant;

private QueryRecordStructure pQueryRecordStructure;

The array contains the constants (i.e., the actual sampled values), while the second variable is the record structure. Methods to retrieve record contents, shown in the uml diagram, have been implemented.



Class: NodeLogicalObjectAttribute

NodeLogicalObjectAttribute is the object charged of representing attributes of a query.
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 it is the only information known at parsing time (pValue still needs to be sampled and its value is unknown). The second variable will be filled only at LLQ execution time, when the value will be physically retrieved.
This class exposes classic getter/setter methods.

Data Structures Computation

The classes described above are used to create the data structures that have to be passed to the QueryAnalyzer class and computed.

Class:Buffer

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 evaluating the if-else-every clause. SamplingData class has been implemented to satisfy this need, as well as to represent all the information about sampling timings.

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

- The if-else-every clause which allows to dynamically select sampling frequency;
  The refresh clause;

- The behaviour that the system is expected to follow in case of unsupported sample rate.

The if-else clause is represented using

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

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

```
private UnsupportedSampleRateOption pBehaviour;

private Node pRefreshClause;
```

where the class UnsupportedSampleRateOption is a Java enum that contains the two currently designed behaviors to be followed. This class obviously exposes a number of getter/setter methods to access all this information.

Class:LLStack

This class represents the tuple <DataCollector,Local Buer, LLQExecutor>, it's internal state is thus the following:

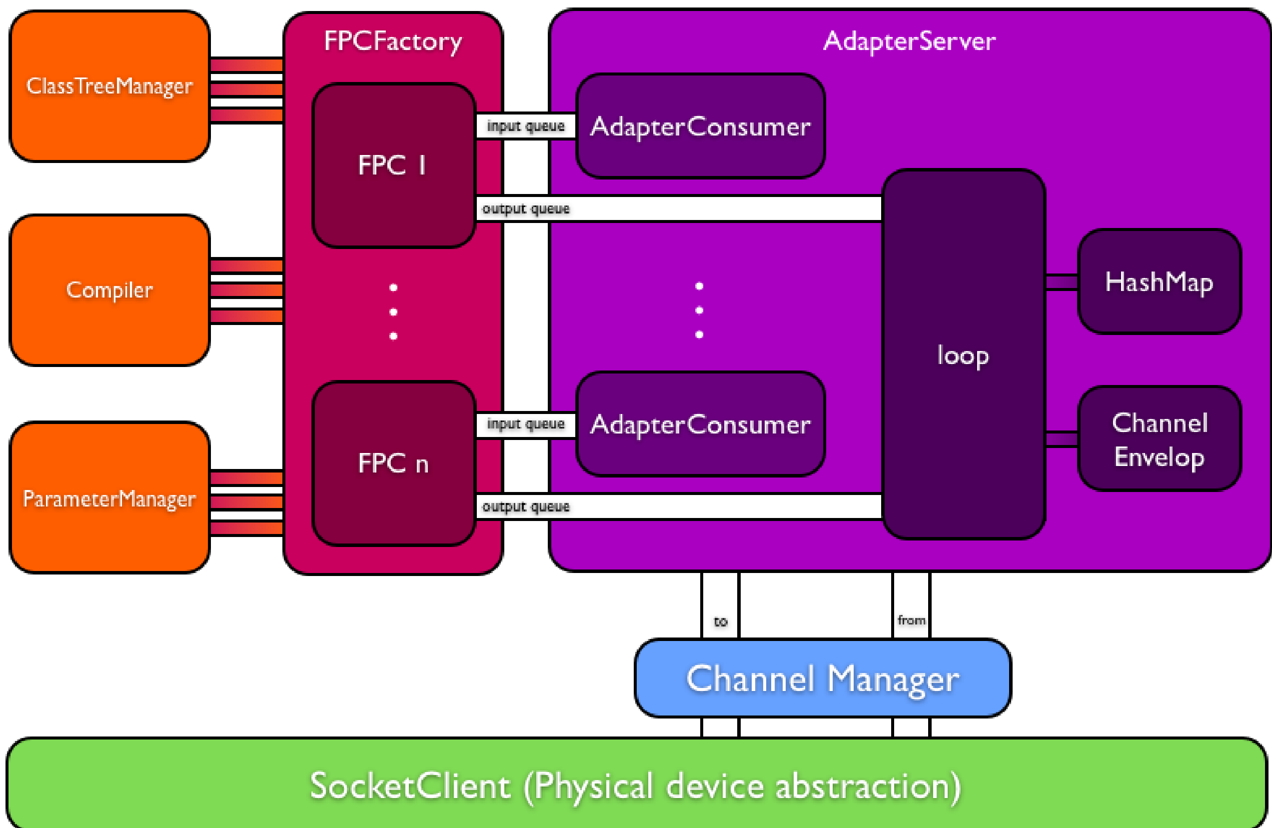
```
private DataCollector pDataCollector;

private Buer pLocalBuer;

private LLExecutor pExecutor;
```

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

Class:QueryInformation

This class represents the information needed by every query to be successfully computed.
It has been chosen to implement this class as Statement class whose information are going to be represented. This choice offers the opportunity to extend this class on need, according to the type of query whose information are going to be represented.

FROM PHYSICAL DEVICE TO FPC



The components

This section explains the behavior of the system when it is started, using a “main” function, implemented to trace the information from the device to its FPC abstraction.

The system uses:

-the AdapterServer that manages the communication between the Channel Manager (which takes data from Socket) and the FPC;

-the FunctionalityProxyComponentFactory, that instances the FPCs starting from a structure written in a xml file;

-the FPCnmarshaller, that joins all the parts of the system.

System working mode

When the system is launched, the first step instantiates the FPCs entities. This work is carried out by the FPC Factory, based on the use of JAXB[06], a group of modules which allows the building of classes starting from an XML representation.

Every FPC can communicate with the Adapter server.

The FPC factory is provided with an ArrayList of FPCs, that can be used during the execution.

The socket receives data from the device and inserts them in an "envelop", using the receiveMessage() method.

```
socket = SocketChannel.getDefaultInstance(); //initializing of socket in main() method

while (fileSize != 0) {

        blobData[i++] = dIn.readByte();

        fileSize--;

        //the content is read byte per byte and is inserted in the ChannelEnvelop

        }

ChannelEnvelop aMessageToSend = new ChannelEnvelop();

aMessageToSend.setMessageId(tDevNumber);

aMessageToSend.setPayload(blobData); //blobdata is put in the ChannelEnvelop
```

After this operation, data are ready for the server, that is waiting with the loop() method.

```
adapter = AdapterServer.getInstance(); //initializing of AdapterServer in main()

ChannelEnvelop message = fromChannell.take(); //selecting of information arriving from the
channel

BlockingQueue<Byte[]> outQueue = outputQueues.get(message.getMessageId());

outQueue.add(message.getPayload());  //sending information to FPC
```

Subsequently data are received by FPC. In the part already explained, the structure of data was formed by "LinkedBlockingQueue[07]" lists.

The next part of project, which collects the data to store them in the database, uses "Pipe" structures.

So the FPC takes data from a LinkedBlockingQueue and put them in a Pipe of Record[05].

In FPCUnmarshal class, there is the useUnmarshallNew(), that uses the method unmarshall(), contained in FPC.

```
public void useUnmarshallNew(FunctionalityProxyComponentFactory factory) {
	List<FunctionalityProxyComponent> fpcs = factory.getFpcArray();

	for (final FunctionalityProxyComponent fpc : fpcs) {
	Thread th = new Thread(new Runnable() {
	@Override

	public void run() {

	LinkedBlockingQueue<Byte[]> receivedData = fpc.getInputData();
	try {
		fpc.unmarshal(receivedData.take());
		} catch (InterruptedException e) {
		e.printStackTrace();
					}
	}
	});
th.start();
}
}
```

For each FPC a thread is started. Each thread waits data and when they arrive, the method unmarshall() starts the conversion.

```
public void unmarshal(Byte[] payload) {

		try {

			DataConverter.fromByteArray(this.pFPCDataStructure, payload);

			Record parMessageToEnqueue=new Record(output);

			parMessageToEnqueue=fillRecord(parMessageToEnqueue,
pFPCDataStructure );				getNewOutputPipe();

			this.outputPipeArrayList.get(0).enqueue(parMessageToEnqueue);


		}catch(IllegalArgumentException e) {		}		}
```

With fromByteArray() the byte array payload is converted to an object, pFPCDataStructure, that is used to form the Record and send it to the output pipe;

We have to pay attention at the FPC constructor, which is called in the Factory.

```
public FunctionalityProxyComponent(int id, String parName, QueryRecordStructure output);
```

The parameter "output" is created in the factory with the method mapThatPojo() and is the parameter used to create the Record type to fill in the pipe.

```
public void mapThatPojo(Object pojo) throws ConstantCreationException{
        Field fields[] = pojo.getClass().getFields();
        for (Field field : fields) {
            String fieldName = field.getName();
            this.addFieldStructure(fieldName, field.getDeclaringClass());
        }
    }
```

fillRecord() inserts information in the Record structure, that is needed to communicate with DataCollector.

```
public Record fillRecord(Record record, Object fpcStruct){
        Constant<?> consta = null;
        Constructor<?> constructor;
        Class tmpClass=fpcStruct.getClass();
        Field[] fields=tmpClass.getDeclaredFields();
        try {
            for (int i=0; i<fields.length; i++){
                Class baseClass=fields[i].getDeclaringClass();
            Class<?> clazz = ConstantFactory.getEquivalentClass(baseClass);
                        try {
            constructor= clazz.getDeclaredConstructor(fields[i].getType());
            consta=(Constant<?>) constructor.newInstance(fields[i].get(fpcStruct));
            } catch (SecurityException e) {
```

```
                e.printStackTrace();
        }       return record; }
```

The method getEquivalentClass() is used to convert the primitive type of information to a complex type that is used by the dataCollector.

getNewOutputPipe() creates a new pipe and inserts it in the outputPipeArrayList.

```
synchronized final public Pipe<Record> getNewOutputPipe(){
        Pipe<Record> pipe = new Pipe<Record>(this.getName() + "_PIPE_" +
(outputPipeArrayList.size()+1));
        pipe.start();
        this.outputPipeArrayList.add(pipe);
        return pipe;
    }
```

Device to FPC transmission structures

Adapter Server: manages the communications between the devices and the Fpc abstraction. It extends Agent, so it has thread features.

The most important method of Adapter Server is loop(), which waits for information. The adapter server uses the LinkedBlockingQueues [07] to communicate with other components.

Adapter Consumer: is contained in the Adapter Server. It creates a ChannelEnvelop for each message arriving from the devices.

Channel Envelop: class that create a wrapper of information (payload of bytes and ID) arriving from the device.

Socket Channel: uses two LinkedBlockingQueues to communicate with Server and with Channel. Socket Channel extends Channel and uses Socket Java class. It creates the Channel Envelop to send it to the Server.

The most important method is loop() which remains in listening status to receive information from the devices.

FROM QUERY TO LOW LEVEL QUERY

This section explains how the Low Level Query environment is linked to the query system of Perla.

For every PerLa query injected into the system, QueryAnalyzer is charged of analyze it to create the Low Level Query structures (LLDataCollectionInfo e LLExecution Info [04]) or the High Level Query Structures, that are not part of this work.

To achieve the creation of this LLQEnvironment class, the Query Analyzer class, that extends the class Component [04] employs a complete set of private methods created ad hoc to analyze queries, each one focusing on a specific aspect .

The constructor receives a string that indicates the name of the component (Channel, Server...) passed to Component class.

```
private QueryAnalyzer(String parName) {
        super(parName);
    }
```

The registerQuery() method is the only public method in the class. After registerQuery() is started, the query is divided in three parts and three method to identify the three query types of Perla (LLQ, HLQ and Set Query).

In this work we point the attention to the createLLStatemenst() method, that creates the environment described above and in [04].

```
public static void registerQuery(Query parQuery) {
        notifyToLogger(Type.Information, "Aggiunta query in corso",
                Verbosity.High);
        createLLStatements(parQuery.getLLStatements());
        createHLStatements(parQuery.getHLStatements());
        createSetParametersEnvironments(parQuery.getSetStatements());
    }
```

LLStatements() method executes some steps, for each identified LL Statement:
-it queries the Local Object Registry to receive a list of FPC able to execute de query;

-builds the Sampling Data Object [04];

-creates the LLQDataCollectionInfo object;

-builds the LLQExecutionInfo[04];

-builds the LLStack, containing a number of DataCollector according to the response of Local Object Recistry. Each query can be executed on one FPC and only one DataCollector for each FPC must be employed.

```
private static void createLLStatements(LLStatementList parList)
```

GetFrequenciesTable() method builds tables with if-every-else clauses.

```
private static SamplingData getFrequenciesTable (LLStatement parStatement)
```

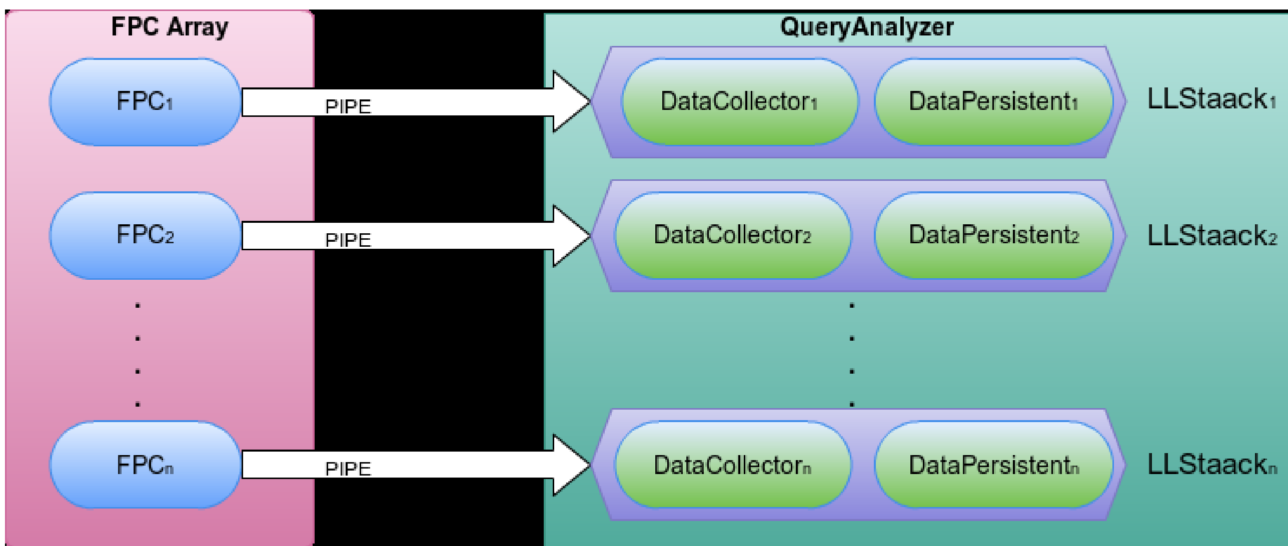
GetWhereCondition() is an internal method to retrieve the where clause.

```
private static Node getWhereCondition(LLStatement parStatement) {
        return parStatement.getSamplingClause().getWhereClause().getCondition();
    }
```

FROM FPC TO QUERY ANALYZER

After completing the initialization step, the FanctionalityProxyComponentFactory generates FPC modules for each device which is connected to the system. The AdapterServer which is bonded to each FPC via outputQueue(outputData) and inputQueue(inputData) through the method AdapterServer.bind(), creates the communication between Channel Manager and FPC structure. During the initialization phase of FPC structures, outputPipeArrayList which contains the pipes going from FPC, will be created and initialized as an empty object. The second pipe will be created and connected to DataCollecter during creation of DataCollector constructor and it will be initialized with pipe waiter object.

After FPC list for the devices has been created, the parsed query will be registered to the QueryAnalyzer which contains methods as createLLStatements and createHLStatements. Because of our query, this process will be followed by createLLStatements(LLStatementList parlist) method to extract the low level statements. In this method we get the FrequenciesTable and WhereCondition from the query with the methods named as getFrequenciesTable(tStatement) and getWhereCondition(tStatement). Moreover the buffer that will be sent to the DataCollector and DataPersistent will be initialized in this step. After allocating and assigning all the variables, new DataCollector object and a new DataPersistent object constructors will be called and placed together into a LLStack which will be added into the stack in an LLEnviroment. The finalization of createLLStatements is the registration of the LLEnviroment into the ActiveSession which will be called in the main function.



```
private static void createLLStatements(LLStatementList parList) {
	SamplingData tTable;
	Node tWhere;
	LLEnvironment tEnvironment;
	LLDataCollectionInfo tDCInfo;
	LLExecutionInfo tEXInfo;
	Buffer tBuffer;

	for (LLStatement tStatement : parList) {
		tEnvironment = new LLEnvironment();
		tTable = getFrequenciesTable(tStatement);
		tWhere = getWhereCondition(tStatement);

		ArrayList<FunctionalityProxyComponent> nodeSet = factory.getFpcArray();

		for (FunctionalityProxyComponent tFPC : nodeSet) {
			tFPC.getClass();
			tBuffer = new Buffer();
			tDCInfo = new LLDataCollectionInfo(tStatement, tFPC, tTable,tWhere, tBuffer);
			tEXInfo = new LLExecutionInfo(tStatement, tBuffer);
			tEXInfo.hashCode();
				try {
					tEnvironment.addStackToEnvironment(new LLStack(
					new DataCollector(tDCInfo), new DataPersistentRecordPolitic
					(tBuffer, tStatement.getTable().getName(), 1)));
				} catch (NullBufferException e) {
							e.printStackTrace();
				} catch (NotValidNameException e) { e.printStackTrace();
				} catch (OutOfBoundsException e) { e.printStackTrace();
				} catch (PerlaConfigurationException e) { e.printStackTrace();
			}
	}
	ActiveSession.addToSession(tEnvironment);
}
notifyToLogger(Type.Information,
"Terminata creazione dei LL Environments", Verbosity.Low);
}
```

The created DataPersistent object that can be DataPersistentRecordPolitics or DataPersistentTimePolitics which depends on the choice of user, will be registered to the AgentRegistry.

```
public LLStack(DataCollector parDataCollector, DataPersistent parExecutor)
	{
		this.pDataCollector = parDataCollector;
		this.pExecutor = parExecutor;
	try {
	AgentRegistry.getDefaultInstance().register(this.pExecutor);
	} catch (Exception e) {
		e.printStackTrace(); }}
```

The method useUnmarshallNew() connects two parts each other and let the data(channelEnvelop) from Channel Manager to corresponding FRC instance.

```
public void useUnmarshallNew(FunctionalityProxyComponentFactory factory) {
        List<FunctionalityProxyComponent> fpcs = factory.getFpcArray();
        for (final FunctionalityProxyComponent fpc : fpcs) {
        Thread th = new Thread(new Runnable() {
        @Override
        public void run() {
        LinkedBlockingQueue<Byte[]> receivedData = fpc.getInputData();
try {
fpc.unmarshal(receivedData.take());
} catch (InterruptedException e) {
e.printStackTrace();
}
}
});
th.start();
}
}
```

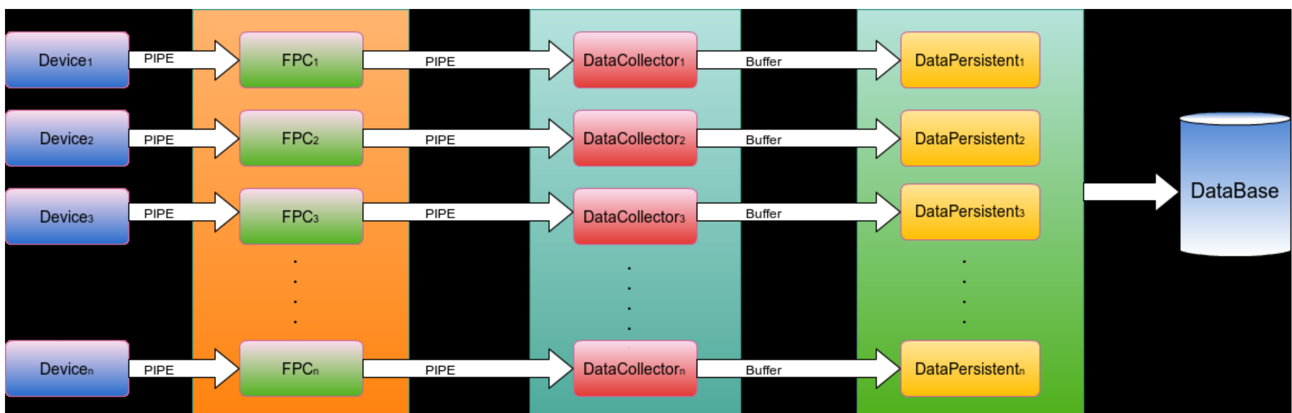
The final part of the main function calls created Environments and from that each LLStack will be obtained as an arraylist and then every LLStack which contains both DataCollector and DataPersistent object will be started. This method triggers the DataCollector thread to start and Agent to lunch DataPersistent object.

```
ArrayList<Environment> parEnviroment = ActiveSession.getEnvironments();
for (final Environment ParEnviroment : parEnviroment)
ParEnviroment.start();
public void start() {
this.pDataCollector.start();
Agent.launch(this.pExecutor);
}
```

In DataCollector, the thread waits for records from the pipe and when a record arrives, it jumps into *performWhere(record)* method in order to be valuated and then the returned record will be put into the outputBuffer which is initialized in QueryAnalyzer class. This buffer is also connected to the DataPersistent object and it is synchronized with that instance which means the waiter inside the DataPersistent object waits for the synchronized arriving buffer. After adding the record to the buffer, it will be transmitted to persistent object and a table will be created according to the given values that are specified inside the configuration file (wsnconfig.conf) such as connectionString, username and password for the database connection.

```
public void loop() throws Exception {
while(true){
actualNumberOfRecord= buff.getBuffer().size();
if(buff.getWaitable() == pWaiter.waitNext() && actualNumberOfRecord >= numberOfRecords){
if(newTable){
try {
System.out.println("creo tabella");
createTable();
```

```
newTable=false;
} catch (SQLException e1) {

e1.printStackTrace();
} catch (ClassNotFoundException e2) {
e2.printStackTrace();
}
}
try {
saveRecords(numberOfRecords);
} catch (OutOfBoundsBufferException e3) {
e3.printStackTrace();
}
}
}
}
```



REFERENCES

[01] 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, IEEE-CS Digital Library 02 Mar. 2011, Vol. 38, n. 2, pp. 478-496 (DOI: 10.1109/TSE.2011.25), (2012);

[02] Fortunato M, Marelli M. - Design of a declarative language for pervasive systems - Master's thesis, Politecnico di Milano, Milano, 2007, http://perlawsn.sourceforge.net/files/documentation/thesis.pdf

[03] Schreiber F. A., Tanca L., Camplani R., Viganò D. - Pushing context-awareness down to the core: more flexibility for the PerLa language- Electronic Proceedings of the 6th PersDB 2012 Workshop (Co-located with VLDB 2012), Istanbul, Aug. 31, pp. 1-6, http://persdb2012.cs.umn.edu/papers/4.Schreiber-PersDB12.pdf , 2012.

[04] Viganò D. - Design and Implementation of Low Level Query Environment for Perla Language – B.Eng. Thesis, Politecnico di Milano, 2010, http://perlawsn.sourceforge.net/files/projects/llq_executor.pdf

[05] Pirotta G., Pisati G., Pozzi D. - Technologies for Information Systems: Project Report - Politecnico di Milano, 2011

[06] http://www.oracle.com/technetwork/articles/javase/index-140168.html

[07] http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/LinkedBlockingQueue.html