TisProject

September 11, 2008

POLITECNICO DI MILANO
FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

PERLA: Functionality Proxy Component

Project of Technologies for Information Systems:

Alessio Pierantozzi
Matricola n. 708539

Roberto Puricelli
Matricola n. 708409

Sergio Vavassori
Matricola n. 711770

ACADEMIC YEAR 2007-2008

Contents

List of Figures

Chapter 1

Introduction

PERLA project is born due to the necessity of offering something more with respect to what there is today in wireless sensor network (WSN) management [3, 6]. Nowadays there are many software implementations that abstract from the single device using an a priori defined representation: the task of implementing the features needed to get the sensor working with the high level view that manages the system is completely left to the sensor's manufacturer.

This approach allows to obtain an homogeneous high level view of the pervasive system, but it had not a big commercial success because the code that manufacturers have to implement is huge and it cannot be easily reused for other management platforms.

PERLA tries to reduce the distance between these two levels, maintaining at the same time the abstraction necessary to avoid the creation of sensor-dependent solutions, and an uniform high level general view of the system.

This part of the project mainly aims at discovering and building some prototypes and at showing that they can work together: thus, these prototypes are not perfect and they must be optimized in the future in order to integrate them in the final work.

In this project the goal of creating a general purpose prototype that can be used not only for a specific case is important. Thus, the features of a sensor are analyzed (section 2) in order to understand what are the useful characteristics that can be used to standardize a language able to describe the physical devices (section 3).

This description is then used to create logical objects that map the physical sensors at an higher level of abstraction. These objects, called *Functional Proxy Component* (FPC), are created dynamically in order to guarantee a *plug and play* enviroment, very useful in a pervasive context.

The process of dynamic generation of the FPC starting from the XML description is provided in section 4. This part is analyzed taking particular attention to the *middleware* that could be used to deal with the sensor. For this purpose an introduction to the *Java Management Extension* (JMX section 4.1) is reported.

Chapter 2

Sensor features

In the context of a pervasive system the definition of a formal representation for each network sensor is important in order to describe the device features. This means that each sensor must be associated to a meta-data representation of its most useful aspects. In this way, an entity (an application or a physical person) that is interested, for example, to query a sensor, can already know what kind of information that sensor could supply. The meta-data representation is a sort of description that gives an a-priori knowledge of the sensor.

The goal, and also one of the difficulties of the project, is to define a general structure of meta-data representation that each sensor would respect. In fact, using a proprietary, non standardized representation of these informations for each sensor, it is not very efficient when these devices are enclosed in the context of a pervasive system. In other words, it is necessary to outline a generic structure that all sensors must follow and that each class of devices can personalize with respect to its own characteristics.

Defining a general set of features that permit to describe every possible sensor in the network is a very difficult task. In fact, there is a large number of little devices used in pervasive networks having different characteristics. For our project purposes not all of these features are interesting. In the description is not important to know the remaining power of a sensor or the distance that each wireless device can cover, while functional aspects, that define the operating mode of each sensor, are much more interesting.

Focusing on the main features of a set of real sensors it is necessary to outline a general structure for sensors representation. This can be done observing some devices in order to find out common features and specific ones.

First of all, it is important to specify that the word “sensor” is used in this context to refer devices capable of sensing lots of parameters as, for example, temperature, humidity, pressure, frequency and others, each one sampled in a different way and discretized with a different Analog-to-Digital Converter.

Sample parameters, as temperature or pressure, that can only be read, are the main features of a sensor. But there are also parameters that can be set, too. This is the case of the sample rate, that can be often modified by an external entity through a special command. Thus, a first distinction within parameters is found.

Another aspect that refers to real sensors, and that must be surely defined in a formal representation,

is the capability of a device. In fact some sensors can periodically sample the value of a parameter in an autonomous manner, while others need to be explicitly invoked for each sampling operation. In the first case, more "intelligent", the sampled value is announced through an event notification every time period, while the second type of sensor follows a request/response paradigm. Every value reported by a sensor is digitally discretized by an ADC and it must be converted to a numeric value useful for human or application analysis.

Moreover each sensor "speaks" its language. This means that every sensor reads or produces a sequence of bytes (interpreted as a message) to receive commands and to report values sensed from the environment. Some of these sensors produce messages according to a Big Endian encoding, while others accept a Little Endian one. It is obvious that this information must be included in a formal description of the sensor, in order to correctly communicate with the devices. This sequence of bytes, and so the message structure, changes from sensor to sensor. Some technologies allow the creation of byte sequences of variable length, for example associating one or more parameter/value pairs, while others are configured to report the actual value of all parameters sensed by the sensor board in each message. Finally, a start and an end flag, or only a message length field, can be used to distinguish each sensor message from the received stream of bytes.

The description reported above focuses on the generic features that can be extracted analyzing sensors. All these aspects need to be expressed in a formal way in order to give an a-piori formal description of the sensor. The next section 3 shows how these characteristics can be formalized using a generic XML representation.

Chapter 3

XML sensor description

The features analyzed in the previous section have been used to create a formal description of the sensor.

There are various ways of describing meta-information, but the most used one that can be adapted to every context, in particular to network contexts, is XML. XML is the best solution also due to its characteristics of writing and validating simplicity. XML representation allows to describe every single sensor with its characteristics and then it is possible to specify validation rules that every XML description must follow. DTDs (Document Type Definition) are less powerful with respect to XSDs (XML-Schema Definition), since they do not support rules concerning data type restrictions, but they are much more simple. These are the reason that leads us to use DTD to describe sensors.

On the basis of the analysis of some real sensors, we identified some significant aspects that are interesting with respect to this project. Generally speaking it is possible to find out some features that are often present in a sensor. In particular it's important to describe *static attributes,* that characterize the device and they don't change during its lifetime (analyzed in detail in the subsection 3.1), *parameter type description* that allows to know the parameters sensed by the device (subsection 3.2), and finally the *message structure* that permits to outline the format of a message (subsection 3.3).

The XML sensor description that follows is intended to be used only as a prototype and it is a first tentative to generically describe peculiar informations characterizing the interesting functional aspects of a sensor. Obviously this formalization must be improved to be used in a real application.

A DTD specification is reported in appendix A, while a sort of XML instance describing a possible sensor is reported in appendix B. This instance is not an example of a real sensor description, but it includes the most important aspects and it gives a baseline over which some modifications can be easily operated.

Moreover this XML instance is also used to test the factory that dynamically produce an output java class representing the Functional Proxy Component (FPC) of the sensor. This part is explained in section 4.

3.1 Static attributes definition

Static attributes of a sensor are the aspects needed to identify and characterize the sensor itself inside a network. The identifier is the feature that defines the sensor identity, for example through a number or

an IP-address. This information is represented in the XML document through the *id* attribute. Also the *encoding* attribute is represented in the XML specification as a static attribute because the encoding type of messages is a property that do not changes during sensor lifetime. In the XML semantics of this attribute could assume the "LE" value to represent Little Endian encoding and the "BE" value to represent Big Endian one.

3.2 Parameter type definition

The parameter type is an important part of the formal definition of the XML document, where the list of parameters that every sensor can sense from the environment is described.

In particular each parameter is identified with the *name* attribute, that is the high level name of the parameter itself (such as "temperature" or "pressure"), the *capability* that is another attribute that can be filled with the "*high*" or "*low*" constants and that indicates the sensing capability of the node. More specifically a low capability requires that every sensing operation is performed only after the receipt of a command from the higher level (request/response paradigm), while a high capability means that the node can sense data autonomously (for example, after the sampling frequency has been set).

As seen in the previous section some parameters can be read or written. Since the communication within the sensor is performed only using messages (section 3.3), a way to distinguish between the different actions must be introduced. It is represented in the XML with *readPermission* and *writePermission* attributes. These are the codes (in general supposed to be expressed in hexadecimal form) that identify the low level operation code used by the device to distinguish the operations. If a parameter can be both written and read, it is associated both to a write low level code and a read low level code. When an operation is requested on a parameter, the right low level code has to be set in order to choose the exact operation to be performed.

Another aspect that formally describes a parameter is the data format, that is expressed in the XML through *type* and measure *unit* attributes of *dataType* element; parameters are also characterized by the *granularity* of the possible assumed values (*continuous* or *discrete*).

To complete the formal description of a parameter it is important to associate to it a *conversionFunction* element that interprets values coming from the sensor and converts them into values that an high level application can understand and verify. There is also a *bounds* element that establishes a minimum and a maximum value that the conversion function can accept to produce a significant value.

3.3 Message structure definition

Message structure is the other important part of the XML document definition that concerns the *structure* each message hat to respect in order to successfully communicate with physical devices. The format of each message must be taken into account to create a message and to send it to the low level device but also to understand and to interpret a message coming from lower level.

In particular three types of messages are defined:

- LeftRightMessage

- LeftSizeMessage
- SizeMessage

Every type of message is represented by a particular sequence of fields that are now explained in detail. In particular, the structure of the messages is reported in the XML, described as the fields included in a message, the length of those fields and some constants.

LeftRightMessage type formalization is characterized by a *LeftDelimiter* and a *RightDelimiter* that mark the message start on the left, and the message termination on the right. Formally each delimiter has a *length* attribute that define the length of the field (expressed in number of bytes) and also a *value* attribute that define the code associated to the delimiter. This code can be written using hexadecimal code or ASCII characters and it has to match with the sequence of bytes used by the devices to delimit messages. The *payload* (described in detail below) is placed between the left and the right delimiters. Also the optional *crc* field can be included. The crc field is defined by a *length* attribute and by *function* attribute that specifies the location of the class that implements the crc function. Figure 3.1 reports an example of *LeftRightMessage* structure.

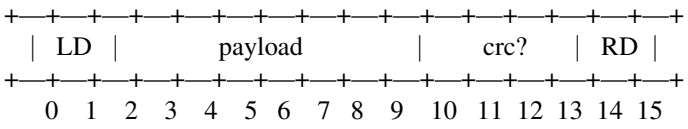

Figure 3.1: LeftRightMessage: example

LeftSizeMessage type is defined as a *LeftDelimiter*, composed in the same way as explained before, followed by a *size* element with a *length* attribute that defines the length of the field expressed in terms of number of bytes. The content of the size field is the binary representation of the size of the entire message. Due to the presence of the *size* field, the right delimiter is not necessary in this kind of messages. As in the previous format the message also contains the payload and the optional *CRC* field as showed in figure 3.2.

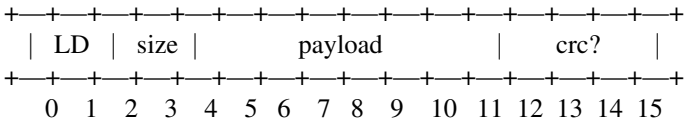

Figure 3.2: LeftSizeMessage

Finally SizeMessage is composed of three fields: the first one contains the message size, the second one is the payload and the third one is the optional *CRC*. Figure 3.3 shows an example in this type of messages.

Two types of payload are defined:

- DynamicPayload
- StaticPayload

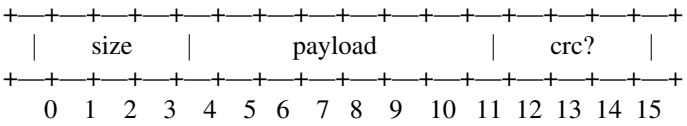

Figure 3.3: SizeMessage

DynamicPayload defines a particular payload containing a variable number of mappings. Each of this mapping elements is characterized by a *refAttribute* that refers to the identifier of the parameter (the low level code associated to the parameter as already told in section 3.2) and by the *valueLength* attribute that specifies the size of the field containing the value associated to that parameter. Every XML instance that uses dynamic payloads must express the maximum number of parameters that can be inserted in a single message (*maxParam* attribute) and the size of the low level names references (*length* attribute). In the described approach, each low level code identifies both a certain parameter and the required operation on that parameter (read/write).
This description of messages contains neither the number of parameters, nor their relative order, but just the maximum number of parameters that can be included in the payload and the size of the field used for the name/value association. This feature actually makes this kind of payload dynamic. Of course the physical device has to be able to deal with such kind of messages and the application that interprets this messages must be implemented according to the characteristics included in the XML (as explained in section 3.3.1).

Figure 3.4 shows a possible dynamic payload that refers to the XML specification reported in appendix B. In particular the eight byte length payload contains two parameters as indicated in maxParam attribute. Each parameter is composed of two bytes (*paramLength* = 2) defining the low level code of the operation on the identified parameter followed by the value associated to that parameter.

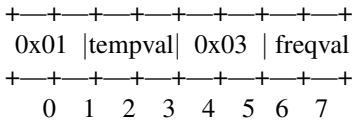

Figure 3.4: First DynamicPayload

Another possible valid instance that respects the specification given in the appendix B contains temperature and pressure readings. In this case the size of the value field for the pressure is three bytes, then the global length of the payload grows to nine bytes. Figure 3.5 shows this payload.

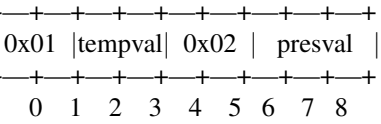

Figure 3.5: Second DynamicPayload

StaticPayload represents the other supported type of payload. In this case values are reported in the message according to the order given in the parameter declaration: the parameters contained in the payload

are always the same and they appears in the same order. For each payload element a mapping containing the value of the referenced parameter and the value field length are specified in the XML.

In the example of static payload, reported in figure 3.6, bytes for temperature, pressure and frequency values are reserved in each message.

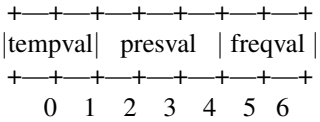

Figure 3.6: StaticPayload

3.3.1 Message management implementation

An application that interacts with a sensor can use the features described above through some static classes that manage sensor messages. In this section a brief description of their implementation is provided.

Any sensor is characterized by a type of message having a default structure as explained before (*LeftRightMessage*, *LeftSizeMessage* or *SizeMessage)* and by a type of payload (*DynamicPayload* or *StaticPayload*). Since the number of these structures is small, it is possible to create classes that permit to manage every possible combination of messages and payloads.

The classes that refer to messages must allow an application that uses them to create a message respecting the format specified in the XML file, and similarly it has to guarantee that every received message can be correctly decoded.

Message.java interface and *Payload.java* interface define the two important methods that represent the basis of every kind of communication between the high level sensor application and the low level device driver.

The first method exposed by the *Message* interface is *createPayload(Map values)* that allows to create a *byte[]* representation of the message given the list of attributes (in terms of name/value pairs). This method is implemented in every message class (*LeftRightMessage.java, LeftSizeMessage.java* and *SizeMesage.java*) in order to manage the differences between the different message types. In particular the *Message* class invokes the method *createPayload(Map values)* exposed by the *Payload* interface (that is implemented in *DynamicPayload.java* and in *StaticPayload.java*): it passes the same parameters list, then it puts the obtained sequence of bytes in the payload field of the message; finally it calculates the message crc, if required.

The reciprocal operation (decoding operation) is supported by the method *getValues(byte [] bytes)* defined both in the *Message* and the *Payload* interfaces. The role of this method is to take the *byte[]* representing the low level message in input, to verify the crc, drop any delimiter and to decode the payload obtaining the attribute pairs (parameter, value).

The conversion of parameter values from human readable values to sensor values and viceversa is performed using the appropriate conversion function (specified in the sensor XML descriptor).

Using this approach the management of the messages is simply performed calling standard methods.

Chapter 4

FPC dynamic generation

The Functional Proxy Component (FPC) is the part of the system developed to represent all the features of the physical sensor at a high level of abstraction.

Before describing the FPC in detail we will introduce how this component is integrated into the system. Assuming that the sensors are well self-described, as already discussed in section 2, the portion of the system included in the dynamic generation process is composed of an *adaptor*, that allows to forward the messages sent from the physical devices to the right recipient, and the *FPC factory*, that is the component able to dynamically generate the *FPC*s (that are the objects charged to map the *sensors*). Once some FPCs are instantiated, a 1:1 mapping between logical objects and physical devices is established. Then, the *Query Executor* can ask the FPC for the values it needs: these queries are converted to low level messages (that can be interpreted directly from the device) and forwarded to the sensors. A simple representation of this process is shown in figure 4.1.

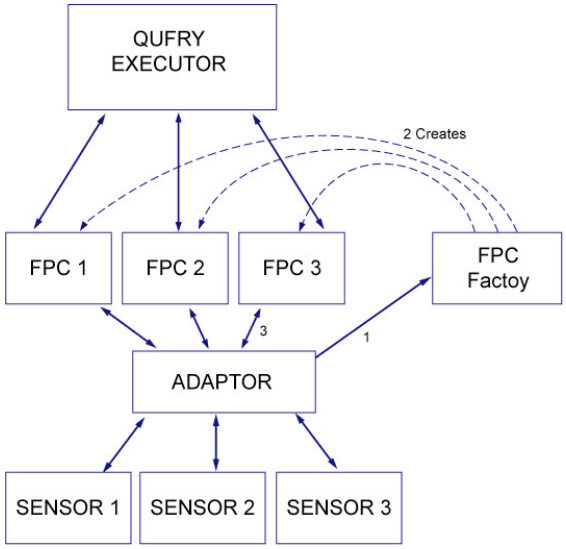

Figure 4.1: FPC system

In the described scenario some critical aspects have to be considered. Firstly, the process of dynamic generation of the FPC component, that is necessary due to the characteristics of pervasive systems, have to be precisely defined. Secondly, the management of the FPC as a single object in a distributed system have to be investigated. In fact, thinking only about the object generation process is not enough: it is also necessary to provide an interface between the FPC and the Query Executor and to allow the lookup of the objects in the system.

The final products of this project propose a solution to these facets. These prototypes are separately presented in the following sections (even if they can be easily combined together). They represent the starting points for the future developments of the project.

4.1 JMX (Java Management Extension)

The first part is concerned with the choice of the middleware that can be used to provide a distributed feature. We analyzed the Java Management Extension (JMX) in order to understand if and how it can fit to our requirements.

JMX technology [8, 1] allows a centralized management of *MBeans* (Managed Beans) which are components that can act as wrappers for resources in a distributed network, through the definition of design patterns and APIs. This functionality is provided by *MBean server*s, which provides registry primitives and exposes interfaces for manipulating the MBeans. These servers are included in *JMX agents* that are Java processes that provide some services to manage a set of MBeans. JMX agents also provide services to create MBean relationships (dynamically loading classes) simple monitoring services, and timers. Agents communication is provided through a set of protocol *adapters* and *connectors* that enable different remote clients to interact with the agent. Protocol adapters and connectors are Java classes, usually MBeans, which can internally map an outside protocol (like HTTP or SNMP) or expose the agent to remote connectivity (like RMI or Jini). The implementation of these components through MBeans and the extremely modular architecture of the server leads to a lightweight solution and allows the server to be easily managed and configured remotely. This 3-layer architecture is represented in figure 4.2.

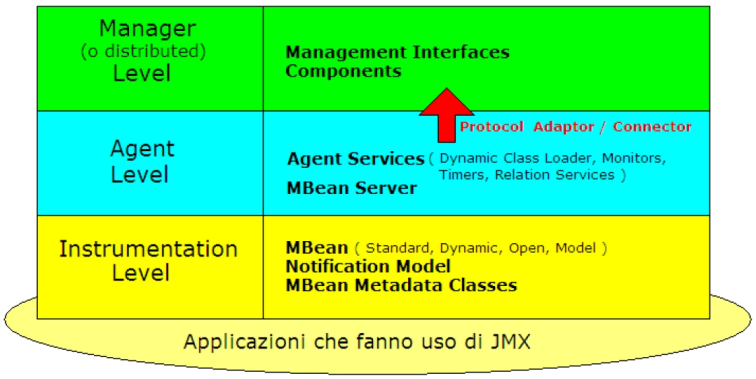

Figure 4.2: JMX architecture

The *distributed layer* is the outermost layer of the JMX architecture. It is responsible of making JMX agents available to the outside world. The *agent layer* provides access to managed resources from the management application. Finally, the *instrumentation layer* includes the MBeans registered in an agent [7]. Each MBean allows the management of a resource through the JMX agent. An MBean looks like a common Java Bean: it is an object that contains some attributes representing its state and it can provide methods to read (*getters*), write (*setters*) or modify (*operations*) this state. All the MBeans should respect certain naming and inheritance standards defined by the JMX specification.

Concerning with the project, *Model MBean* is a particularly interesting kind of bean. The main feature of this object is that developers do not have to write a MBean class. Model MBeans, in fact, can be instantiated in the MBean server and configured by a user to manage any resource only describing its interface through an *MBeanInfo* Object. Moreover, a Model MBean provides several features that make it more robust than a common MBean, in terms of persistence, attribute value caching, etc.

4.1.1 Advantages using JMX

JMX technology presents some benefits concerning the aims of the project. First of all it is a standard Java-based solution, that means complete portability on different platforms. JMX is also included, as already told before, in Java Standard Edition, thus there are no needs to include external libraries in order to execute the application on a machine. Moreover, JMX is already implemented, tested and widely used. For this reason it is surely a better solution than trying to build and test an application dependent middleware, that is difficult to correctly develop and manage. JMX provides the capability of establishing communication with any protocol (such as HTTP) and connectivity with any other transport protocol (such as Java RMI).

Some of the features needed at the middleware level of PERLA project are already implemented. For example the dynamism of the Model MBean fits with the problem of the run-time definition of the management interface. In fact the interface of the FPC can be described outside the MBean. JMX also provides a notification or timing system: this feature can be used with sensors that are not enough sophisticated to manage alert messages or sampling. Finally, it is possible to query the JMX server on the needed attributes and it is possible to retrieve only the interesting logical devices, with respect to how they were registered.

The only limit related to this technology is the fully centralized management of the MBeans: each MBean can be registered only in a single server. The use of Jini technology for the lookup service of the servers can be a practical solution to this problem. Anyway the direct lookup of the single MBean (without the knowledge of the servers) is difficult. Thus, for example, if you want to lookup all the sensors that manage the temperature parameter, you should firstly find all the servers in the system, using Jini services and then query each server to discover all the FPCs that manage temperature sensors.

4.2 General architecture

The scenario this project was implemented for deals with sensors that are supposed to be connected to an *ARM* architecture processor, one of the widely used technologies in embedded systems due to its power saving features. Of course this advantage is compensated with limited resource trade-off. For these reasons

resource limitations must be taken into account while developing sensor-side software. In fact it's not possible to load a standard java virtual machine without overloading all the resources. A possible solution is the use of *GCJ* [5], a free-software Java compiler, since there are no SUN-certified Java 5.0 virtual machines for the ARM architecture. Then the decision of writing the code in Java 1.4 should reduce again the needed resources.

There are no particular constraints related to the *server-side* software, since it is supposed to be executed on standard processors. The main purpose of this part of the system is to dynamically create the FPC: This is still done for performances improvement.

The last part of the system is composed of the clients that need to query the MBeans: in figure 4.1, they are represented by the Query Executor.

The best way to explain how the system works is to take a look at figure 4.3.

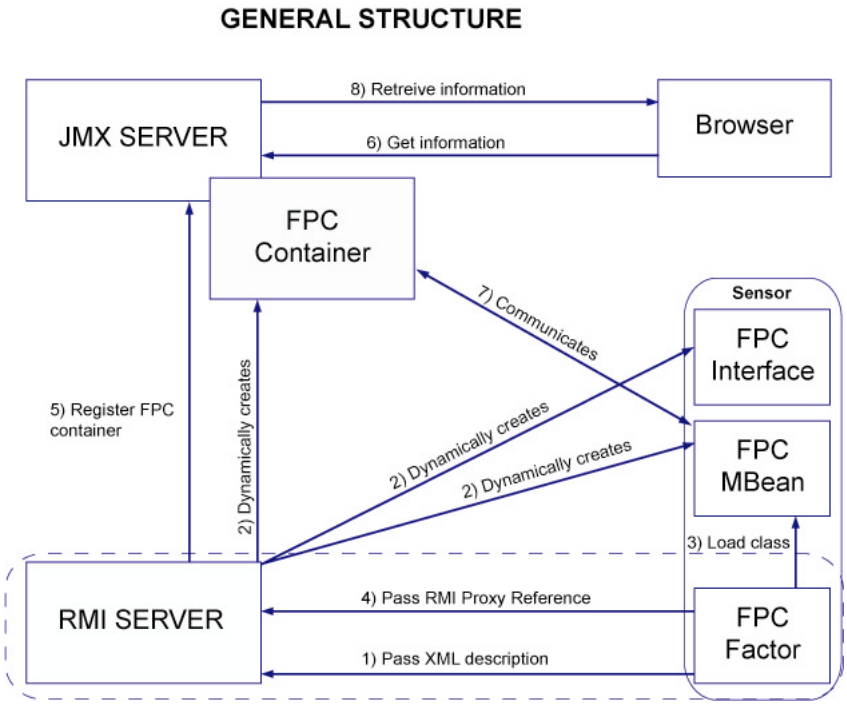

Figure 4.3: General architecture of the system

The *FPC Factory* receives the *XML description* from the sensor and forwards it to the *RMIServer* (1). Then, the *RMIServer* translates the received description and generates the Java code of the *FPC interface*, the *FPC MBean* (treated as a Model MBean as explained above) and the *FPC Container* (2). This last object implements the same interface as the *FPC MBean* and it is the one that is really registered in the MBean Server. It also contains the *RMI proxy reference* of the FPC MBean. This is necessary due to the lack of implementation in the current available version of JMX. In fact in the JMX specification it's included the possibility of setting an "*RMIReference*" as a managed resource, but unfortunately it's not yet available in the current implementation. Using this "trick" the described problem is avoided and the FPC container, even if locally registered, can communicate with the sensor side, forwarding the method calls recursively

invoking the same methods on the proxy reference. In the *FPC Container* each method is implemented only invoking the method with the same name on the *FPC MBean*.

The compilation of the code is completely performed on server side but it is different for the sensor side (compiled with Java 1.4) and the server side (compiled with Java 5.0). In fact, with the compilation in Java 5.0, the *skeleton* (needed for the communication) of a *Remote* class is already included in the java class file. The *.class* files related to the sensor are then returned to the FPC Factory, that loads the MBean (3) and passes its proxy reference to the server (4). At this point the RMI Server can register the FPC Container (and indirectly the FPC MBean) into the server (5). Finally, a browser can lookup the sensor FPC and invoke some methods (6). The communication is automatically established (7) and the result is that the state values stored in the sensor-side are correctly returned to the server-side (8).

4.2.1 Problems

The above described architecture has been tested and it is working on a standard virtual machine. But there is a non-trivial problem with the *GCJ* virtual machine. The exportation of the RMI proxy is not correctly performed when an IP address is passed as Virtual Machine parameter "*-Djava.rmi.server.hostname*": the constant value *localhost* is always used, instead of the received value. This makes the communication impossible. Unfortunately it is a known problem of the virtual machine and the only available solution (not yet confirmed by the developers of the compiler) is to recompile all the GCJ sources on every sensor [2, 4].

An alternative virtual machine should be found due to this issue: in fact the CGJ development seems a bit stuck in the last period and a new release that can solve the problem can not be expected soon.

Another problem is the *cost of communication* since the radio channel is very expensive in terms of consumed energy and the architecture requires a radio communication for every request. Thus, the communication duration is certainly a parameter to minimize and, since JMX cannot directly stay on sensor-side it is not possible to use all the features provided by JMX.

4.3 The Functional Proxy Component

The second prototype presented in this project is the creation of the Functional Proxy Component starting from the XML description of a sensor. This represents the starting point of the development of this component of the system. This section will introduce the ideas that are at the basis of this study without entering in the implementation details.

The FPC is not so easy to define. There are a lot of features to take into account while developing it. It has to fit in the general structure of the system and the interfaces with the lower and upper levels are critical to define. As shown in the figure 4.1 the FPC is connected with the *adapter*, that forwards the correct messages to and from the correspondent physical sensor, and with the *Query Executor*. The first connection can be done using two *Pipe*s, that are unidirectional communication channels that allow to store the messages and to read them following the FIFO order.

The interface with the Query Executor is instead more complicated. In fact, more queries, involving the same physical sensor, can be launched at the same time. Thus, a sort of scheduler is needed in order to

manage this concurrency. Moreover, the Query Executor is implemented to optimize the query execution: it asks the FPC for a set of parameters and it uses the result fill a data structure that makes possible the query execution.

The main methods exposed to the upper level are *getParameters*, which takes as input a list of parameters names and returns a list of values corresponding to the request and *setParameters*, that converts a Map (parameter name - value) to a request for the sensor.

The parameters of the sensor are managed with the appropriate *ParameterMonitor*. This class is a monitor that allows multiple parallel readers and serializes multiple writers on the value. This feature is very important because it is necessary to synchronize all the accesses to the variable in order to guarantee consistency, integrity and availability of the correct value. This concept is also an argument of discussion: what is a *correct value*? The answer to this question can be trivial: it is the current value of the parameter registered from the sensor. This is, of course, difficult to implement.

An approximation can be done at this level: a value is correct if it was updated at a timestamp enough closed to the request timestamp. That's why a *lease* value is added to each parameter. This value is used by the *getParameters* method in order to fill the fields to return to the Query Executor . If the lease is expired a new thread (*ParameterThread*) is created with the purpose of waiting until the parameter is updated and then fill the return value. These updates are required from the FPC sending messages on the *output* Pipe. This process is done trying to minimize the number of messages sent with respect to the specification of the XML description (section 3.3).

As said before, the incoming messages from the adapter are coming on the *input Pipe*. This Pipe is managed using an apposite thread (*PipeReaderThread*), that waits for new messages on the Pipe. When a message is received, the PipeReaderThread just interprets it and updates the corresponding value. The messages are managed using the static class that were discussed in section 3.3.1.

Chapter 5

Conclusion

The main purpose of this project was to demonstrate the feasibility of the dynamic generation of an FPC and to study how it can be included in a distributed system in order to manage queries on a pervasive system. The FPC is a class able to abstract a physical sensor to a logical object that can represent all its features.

The most critical aspects we dealt with are reported in the following list:

1. managing of the interaction among the components of the system in a distributed middleware
2. representing in the most general way the features of a real sensor through an *XML description*
3. analyzing the requirements of the scheduler required to allow the execution of multiple queries on the same FPC

The two prototypes created show that the previous features can be provided and they propose some ideas to deal with those critical aspects.

Appendix A

Appendix: Document Type Definition (DTD)

```
<!ELEMENT sensor (parameters, message)>
<!ELEMENT parameters (parameter+)>
<!ELEMENT parameter ((readPermission | writePermission |
 (readPermission, writePermission)),bounds, granularity,
 dataType, conversionFunction)>
<!ELEMENT readPermission EMPTY>
<!ELEMENT writePermission EMPTY>
<!ELEMENT bounds EMPTY>
<!ELEMENT granularity EMPTY>
<!ELEMENT dataType EMPTY>
<!ELEMENT conversionFunction EMPTY>
<!ELEMENT message (leftRightMessage | leftSizeMessage | sizeMessage)>
<!ELEMENT leftRightMessage (leftDelimiter, payload, crc?, rightDelimiter)>
<!ELEMENT leftSizeMessage (leftDelimiter, size, payload, crc?)>
<!ELEMENT sizeMessage (size, payload, crc?)>
<!ELEMENT leftDelimiter EMPTY>
<!ELEMENT payload (staticPayload | dynamicPayload)>
<!ELEMENT crc EMPTY>
<!ELEMENT rightDelimiter EMPTY>
<!ELEMENT size EMPTY>
<!ELEMENT staticPayload (mapping+)>
<!ELEMENT dynamicPayload (mapping+)>
<!ELEMENT mapping EMPTY>
<!ATTLIST sensor
```

```
          id   ID   #REQUIRED
          encoding   (LE | BE)   "LE">
<!ATTLIST parameter
          name   ID   #REQUIRED
          capabilities   (low | medium | high)   "low">
<!ATTLIST readPermission   code   ID   #REQUIRED>
<!ATTLIST writePermission   code   ID   #REQUIRED>
<!ATTLIST bounds
          down   CDATA   #REQUIRED
          up   CDATA   #REQUIRED>
<!ATTLIST granularity   gran   (discrete | continuous)   "discrete">
<!ATTLIST dataType
          type   CDATA   #REQUIRED
          unit   CDATA   #REQUIRED>
<!ATTLIST conversionFunction   class   CDATA   #REQUIRED>
<!ATTLIST leftDelimiter
          length   CDATA   #REQUIRED
          value   CDATA   #REQUIRED>
<!ATTLIST dynamicPayload
          paramLength   CDATA   #REQUIRED
          maxParam   CDATA   #IMPLIED>
<!ATTLIST crc
          length   CDATA   #REQUIRED
          function   CDATA   #REQUIRED>
<!ATTLIST rightDelimiter
          length   CDATA   #REQUIRED
          value   CDATA   #REQUIRED>
<!ATTLIST size   length   CDATA   #REQUIRED>
<!ATTLIST mapping
          refAttribute   IDREF   #REQUIRED
          valueLength   CDATA   #REQUIRED>
```

Appendix B

Appendix: XML Example

```
<sensor id="012345" encoding="LE">
   <parameters>
      <parameter name="temperature" capabilities="low">
         <readPermission code="0x01" />
         <bounds down="0" up="100"/>
         <granularity gran="discrete"/>
         <dataType type="int" unit="it.polimi.tis.converter.Celsius"/>
         <conversionFunction class="org.dei.perla.sensor.test.TestConverter"/>
      </parameter>
      <parameter name="pressure" capabilities="low">
         <readPermission code="0x02" />
         <bounds down="500" up="1000"/>
         <granularity gran="discrete"/>
         <dataType type="int" unit="Pascal"/>
         <conversionFunction class="org.dei.perla.sensor.test.TestConverter"/>
      </parameter>
      <parameter name="frequency" capabilities="low">
         <readPermission code="0x03" />
         <writePermission code="0x04" />
         <bounds down="5000" up="10000"/>
         <granularity gran="discrete"/>
         <dataType type="int" unit="Hertz"/>
         <conversionFunction class="org.dei.perla.sensor.test.TestConverter"/>
      </parameter>
   </parameters>
   <message>
      <leftRightMessage>
```

```
            <leftDelimiter length="2" value="LD"/>
            <payload>
               <dynamicPayload paramLength="2" maxParam="2">
                  <mapping refAttribute="temperature" valueLength="2"/>
                  <mapping refAttribute="pressure" valueLength="3"/>
                  <mapping refAttribute="frequency" valueLength="2"/>
               </dynamicPayload>
            </payload>
            <crc length="4" function="crcSum.class"/>
            <rightDelimiter length="2" value="RD"/>
         </leftRightMessage>
      </message>
   </sensor>
```

Bibliography

[1] M.B. Whipple B.G. Sullins. *JMX in Action*, 2003.

[2] GCC bugzilla. *GCJ bug description nÂ°29501*, October 2006. http://gcc.gnu.org/bugzilla/show bug.cgi.

[3] M. Fortunato M. Marelli F. Pacifici F.A. Schreiber, R. Camplani. *PERLA: a Data Language for Pervasive Systems*, 2007. Politecnico di Milano, Dipartimento di Elettronica e Informazione, Milano, Italy.

[4] forum GCJ. *GCJ bug proposed solution*, January 2006. http://lists.gnu.org/archive/html/classpath/2006-01/msg00098.html.

[5] GCC group. *The GNU Compiler for the Java Programming Language*, August 2007. http://gcc.gnu.org/java/.

[6] M Marelli M.Fortunato. *Design of a declarative language for pervasive systems*, September 2007. Master's thesis, Politecnico di Milano.

[7] SUN microsystems. *Getting Started with Java Management Extensions (JMX): Developing Management and Monitoring Solutions*, January 2004. http://java.sun.com/developer/technicalArticles/J2SE/jmx.html.

[8] SUN microsystems. *Java Management Extensions (JMX) Technology homepage*, 2004. http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/.