PEDiGREE

PErvasive Database GRoup of EnginEers

The PerLa System

Fabio Alberto Schreiber

Emanuele Panigati

Guido Rota

Francesco Maria Filipazzi

Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO DI MILANO

Introduction

There are many real world 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. *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. 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 analyser 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 remainder of this document contains a complete description of the entire PerLa System. The PerLa Query language is described in Section 2. An in-depth overview of the Logical Object component and its supporting modules can be found in Section 3. Sections 4, 5 and 6 describe the implementation of three extension to the PerLa Middleware, namely an HTTP communication library, a TinyOS integration module and a driver for the Cassandra NoSQL database system. Finally, a summary of all ongoing projects and future works is presented in Section 7.

1 System Overview

The architecture of the PerLa system (Schreiber et al. [2012a]) can be partitioned into three different software layers with clearly defined interfaces: a Hardware Abstraction Layer, a Query Execution service and a Context Management System. Each of these layers builds on the previous ones to provide additional functionalities to the entire framework.

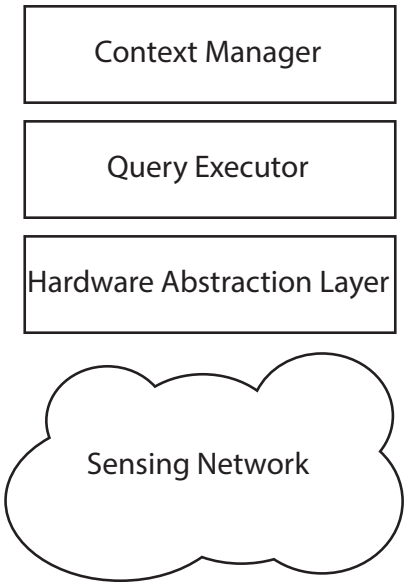


Figure 1: High level overview of the main components

Hardware Abstraction Layer The Hardware Abstraction Layer (HAL) is responsible for managing the lifecycle of all devices connected to the PerLa framework, and for providing a uniform API to interact with them. Its design revolves around the Functionality Proxy Component (FPC), a self-contained proxy object that embeds all the logic required to communicate with a single remote device.

Query Executor The component responsible for executing queries written in PerLa. The PerLa language is a SQL-like query language that allows end users and high level applications to gather and process information without any knowledge of the underlying pervasive system.

Context Manager The Context Management layer is an extension to the PerLa framework designed to simplify the development and deployment of context-aware applications based on distributed sensing networks. It consists of a Context Language and a Context Management component, both of which interface directly with the Query Executor layer. The Context Language enables PerLa users to define the context-driven behaviour of a network of distributed computing nodes; context is represented using the Context Dimension Tree model (CDT, Bolchini et al. [2009]), while context-dependent actions are specified using PerLa Queries. The Context Management component is responsible for populating the CDT with data collected from the sensing network, as well as tailoring and performing the related PerLa query associated with the active context.

2 The PerLa Query Language

PerLa is a framework to configure and manage data in pervasive systems and, in particular, in wireless sensor networks. The language provides three different query types:

High Level Queries (HLQs) Define the high level elaboration involving data streams coming from multiple nodes. Currently these operations are executed by a DBMSs , like *Cassandra* or *MySQL*, that is supported by the PerLa data wrappers. In future a High Level Query processing module will be developed directly integrated in PerLa. In any case, these queries are associated with the *high level support layer* and run on the central server.

Low Level Queries (LLQs) Define the behavior of every single node or of a homogeneous group of nodes. In particular, LLQs specify the data selection criteria and the sampling frequency of each node or set of nodes. These queries are associated with the *low level support layer*.

Actuation Queries (AQs) These queries run on devices that have capabilities of performing some kind of action; the content of these queries is a set of commands that activate some kind of actuator.

In PerLa two different data structures are provided:

Snapshot sets of record generated during a specified time period.

Streams unbounded tables designed to be mostly output structures.

Both structures are characterized by an Id, a type and a timestamp. Low Level Queries need a local buffer since they retrieve data from sensors, push them into the buffer, take values from the buffer and insert the obtained records into the output data structures, so preventing useless power consumption due to unnecessary frequent transmissions. Garbage collection is needed to keep the buffer clean and up-to-date.

Example of a SNAPSHOT query This query gets data related to the last time a vehicle was used in a system of car rental management.

```
1 CREATE SNAPSHOT LastStart (greenBox_id String, timeMinutes Integer,
      GPSData position)
2     WITH DURATION 2h AS LOW
3 SELECT greenBox_id, timeMinutes, position
4 HAVING timing = MAX (timing, 2h)
5 SAMPLING ON EVENT ChangeVehicleStatus
6 WHERE ed_status = 0
```

Example of a STREAM query This query lists all the meteorologic changes toward bad weather. "Ice" is considered when the temperature is less or equal than 4°C, thus roads start to become slippery.

```
1 CREATE STREAM WeatherChange (position, GPS_Data, climate String) AS LOW:
2 EVERY 10 m
3 SELECT position, climate
4 SAMPLING
5 ON EVENT WeatherChanged
6 WHERE climate = "Rain" OR climate = "Ice" OR climate = "Snow" OR
      climate = "Fog"
7 REFRESH EVERY 5 m
```

Each PerLa query can be composed of several different clauses.

Select clause and data management The *SELECT* clause is the most important clause of the query, since it specifies which are the data to be retrieved from the sources. The Select clause can be associated with a "having clause" that works with both aggregates and single values. The SELECT clause is evaluated with a frequency defined in the EVERY clause of the insertion query, as described before. The idea behind the SELECT clause is the same as in standard SQL: they both have the role of specifying how data have to be collected from data sources (an unbounded buffer in our case; the tables in SQL) and manipulated to produce some records as result.

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

Execution condition The *EXECUTE IF* clause allows the middleware to understand which are the node(s) on which the query must be run: the predicates contained in the EXECUTE IF clause state which boolean condition a node must satisfy in order to be considered as a candidate data source for the query. If the node does not provide all the necessary attributes, the query will not be run on it. In the end, the query will be run on all and only the nodes that can give a correct answer to. The middleware component responsible for the evaluation of the execution condition is the FPC Registry. Both functional and non-functional attributes can be taken into account. A *REFRESH* clause can be appended to the

EXECUTE IF one in order to indicate if and when the system should revaluate the condition to update the list of logical objects executing the query.

TERMINATE AFTER is the termination statement that establishes when the query has to stop its execution. It is an optional clause that can be used to terminate the execution of a query. This behavior may be useful to perform a one-shot query, as shown in previous example, or when the monitoring period is known a priori.

```
1 CREATE OUTPUT STREAM Table (rfid STRING, counter INTEGER) AS LOW:
2 EVERY 10 min
3 SELECT lastReaderId, COUNT(*, 10 min)
4 SAMPLING
5 ON EVENT lastReaderChanged
6 EXECUTE IF ID=[tag]
7 TERMINATE AFTER 1 SELECTIONS
```

Sampling clause With the term "sampling" we mean that one or more attributes of the logical object are read and the obtained record is inserted into a local buffer, if it satisfies some conditions. Note that the reading of an attribute can imply a sensor sampling if the attribute is dynamic.

The SAMPLING clause specifies how to collect data and how and when a sampling operation should be performed. Both event-based and time-based sampling semantics are supported. The first one is obtained writing an ON_EVENT clause and forces a sampling whenever an event is raised by the FPC; the second one is obtained writing an IF_EVERY clause and forces the sampling to be periodically executed with a certain frequency, which can be parametrically specified.

When the time based sampling is used, other two clauses can be specified in the statement. The first one allows to define the behavior of the node if the required sampling rate is too fast and then unsupported. The "DO NOT SAMPLE" option indicates that the query execution should be suspended until the required sampling rate becomes acceptable again. Otherwise the "SLOW DOWN" option allows keeping on the query execution, but with an automatic reduction of the sampling

rate. If the clause is not specified, the option "DO NOT SAMPLE" is used as the default one. The second clause is the REFRESH one and it allows the user to specify if the sample rate is fixed for the whole query life or if it should be periodically revaluated.

The SAMPLING clause can be completed appending a WHERE clause, that allows the definition of a filtering condition. If specified, this condition is evaluated whenever a record is produced, before the insertion into the local buffer. If the condition is not satisfied, the record is immediately rejected without inserting it into the buffer.

In order to execute a query the following middleware components are needed:

HLQ EXECUTOR Is currently a wrapper for external data management systems, and it is interfaced with Cassandra and MySql.

LLQ EXECUTOR A LLQ Executor is created for each FPC involved to operate on the related node(s)

2.1 Low Level Query Executor

Although users can interact with a sensing node using the programmatic FPC interface, the easiest way to extract information from a sensing network managed by PerLa is through the use of the PerLa Query Language. PerLa queries are analyzed and executed by the Query Executor, which operates in tight cooperation with the Hardware Abstraction Level.

The first software component in this layer is the **Parser**, which is tasked with translating textual PerLa queries into an appropriate Java representation. It receives the queries directly from the user (to whom it reports the execution results) and sends them to the Query Analyzer. Parsed queries are then transferred to the **Query Analyzer**, where they are inspected to create a feasible execution plan. It is responsible for the creation of the HLQ Executor and the LLQ Executor for the related queries.

The execution process begins by querying the FPC Registry for a collection of FPC objects that can be used as data sources for the query. This selection may be reevaluated periodically if the set of devices which satisfy the execution

condition is likely to change while the query still running. Each FPC selected in the previous step is then connected to a query management stack, composed of a Sampling Manager and a Data manager. These two components are used to schedule sampling tasks on the remote device, and to perform data manipulation operations on the sampled records respectively.

3 Hardware Abstraction Layer

This section contains a comprehensive description of the PerLa Middleware components that form the Hardware Abstraction Layer.

3.1 Functionality Proxy Component – FPC

The PerLa Middleware is a collection of software units designed to support the execution of PerLa queries. Its design revolves around the *Functionality Proxy Component* (FPC), a proxy object which acts as a decoupling element between sensing nodes and middleware users. Essentially, the PerLa middleware constitutes the software environment needed to manage the lifecycle of a set of FPCs.

Overview Functionality Proxy Components are dynamically created by a Factory module, which assembles new FPC objects from a formal description of the node being registered into the middleware.

The most prominent trait of the FPC is its hardware-agnostic interface, a uniform API that provides a consistent method of interaction with all the heterogeneous devices found in a (wireless) sensing network. Viewed through the abstraction provided by an FPC, each device is a collection of attribute whose values can be queried or altered using two primitive operations, respectively called *get()* and *set()*. The usage of these two basic commands neither requires knowledge of the sensing network, nor of the device that will ultimately perform them. Attribute values are produced or consumed by a computing node. Inside the PerLa middleware, attributes are characterized by a name and a data type; this

meta-information enables PerLa users to correctly address a single specific piece of information. PerLa currently supports the following data types:

INTEGER Integer numbers

FLOAT Fractional values in floating point representation

STRING Sequence of characters

BOOLEAN Logical value (true or false)

ID PerLa device identifier

TIMESTAMP Identifier of a specific moment in time

The FPC interface The concept of *Device Attribute* is mainly employed to correlate middleware components with the information that they provide access to.

The FPC interface is entirely defined in terms of operations on attributes:

Task get(Attributes, TaskHandler) Retrieves the value of the specified attributes

Task get(Attributes, Period, TaskHandler) Periodically retrieves the value of the specified attributes

Task set(Attributes, Values, TaskHandler) Sets the value of the specified attributes

As explained in previous paragraphs, accessing information through the primitive operations provided by a FPC do not require any knowledge of the sensing network; the user only needs to specify which attributes must be retrieved or modified.

All the operations performed by the FPCs are asynchronous and non-blocking. The immediate result of all FPC methods' invocations is a *Task*, an object whose function is to allow control over the requested asynchronous computation (e.g., query for completion, cancel the operation, ...). The output of the operation, once ready, is delivered with an asynchronous invocation of the *TaskHandler* provided at request time. This operating principle is the high-level equivalent of micro-controller hardware interrupts: users define an interrupt source and an interrupt handler, then the micro-controller asynchronously invokes the callback handler

whenever a new event is generated.

The non-blocking FPC interface is a feature introduced with the new PerLa middleware architecture that aims at improving reaction times and increase the overall system scalability. The reactive model fostered by this new design addresses two flaws of the previous implementation, namely the excessive number of threads required to parallelize several synchronous I/O operations, and the loss of performance due to frequent context switches imposed by the former thread-based model. It is worth noting that this non-blocking interface design is extended to all software components of the PerLa Middleware.

Operation management and scheduling The current FPC implementation supports the execution of multiple concurrent *get()* and *set()* operations. The actual scheduling of processing tasks on the remote device varies according to runtime conditions, as it depends on the number and type of requested operations, as well as the capabilities of the remote computing device itself.

The FPC may, for example, schedule a single periodic sampling operation on the sensing node, and fan-out its outcome to several consumers, potentially re-sampling the data stream if the requested sampling rates are different. Otherwise, in case of several non-periodic (one-off) *get()* operations, the execution may be performed sequentially.

The FPC is also responsible for simulating certain types of operations, when these are not provided by the remote sensing device.

Internal FPC Components The FPC is composed of several independent software units, each of which is responsible for managing a single aspect of the interaction with the remote device. This new modular design improves on the previous monolithic architecture by promoting component reusability and greater decoupling between units with separate concerns, and provides a cleaner interface better suited for expandability.

Every FPC object is composed of the following units:

CHANNEL A software component capable of performing I/O operations. Channels are responsible for managing the communication between the PerLa

Middleware and the remote devices.

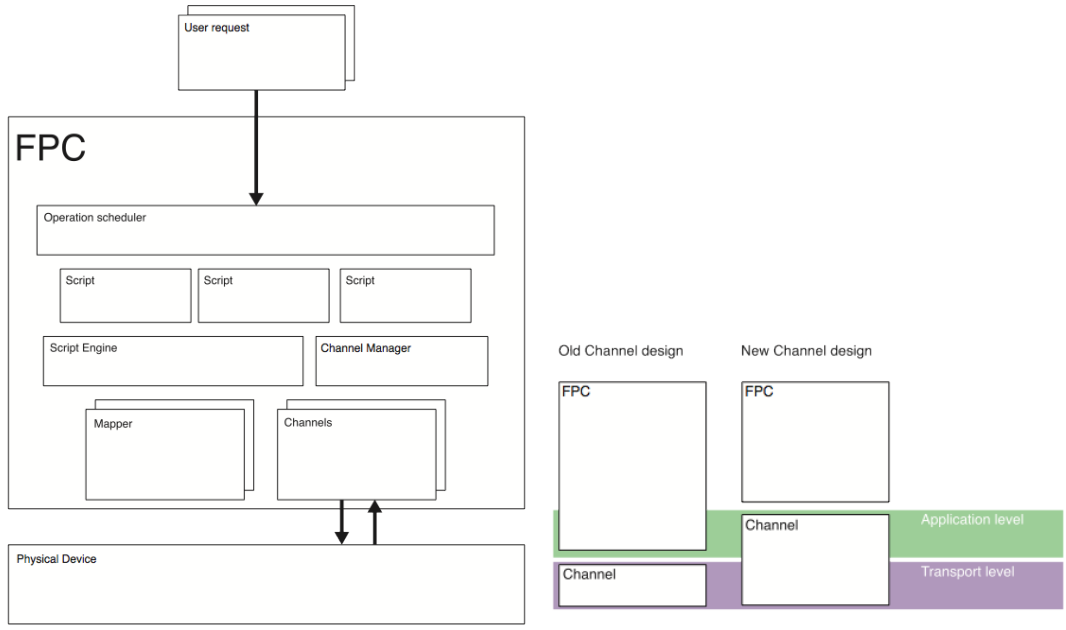
MAPPER Data marshaller/unmarshaller. *Message Mappers* allow the FPC to interpret byte streams received from a Channel and to serialise Java object prior to transmission.

CHANNEL MANAGER The component which manages all the *Channels* used by the FPC. The *Channel Manager* is a fundamental part of the PerLa architecture, as it is responsible for dispatching asynchronous events and data towards the appropriate consumer.

SCRIPT ENGINE A simple engine for running small scripts, used by the FPC to dynamically bind high level data requests to native processing tasks run on the remote device.

OPERATION SCHEDULER Manages the scheduling of processing task on the remote device.

Additional details on the internal software architecture of the Channel, Mapper and Script Engine components are available in the following.



(a) FPCs reenginered architecture (b) Channel reenginered structure

Figure 2: New FPC and Channel design

```
<?xml version="1.0" encoding="UTF-8"?>
<device type="test_descriptor"
        xmlns="http://perla.dei.org/device"
        xmlns:http="http://perla.dei.org/channel/http"
        xmlns:ser="http://perla.dei.org/channel/serial"
        xmlns:ue="http://perla.dei.org/fpc/message/urlencoded"
        xmlns:js="http://perla.dei.org/fpc/message/json">
```

Listing 1: Device description header

As previously mentioned, FPCs are dynamically created at runtime by an object factory, called *FpcFactory*. This component is responsible for the instantiation and assembly of all the constituent part of an FPC.

The starting point for the creation of a new FPC is the Device Descriptor, an XML document which contains a machine parseable description of a single computing device of the sensing network. Device Descriptors are organized in different sections; each section defines the behaviour of a specific aspect of the FPC being created.

The first part of a Device Descriptor contains a textual description of the physical device (type attribute), and a series of XML namespaces. The XML namespaces have a twofold function in the PerLa middleware: first and foremost, namespaces are used to disambiguate element and attribute names; then namespaces are used by the *FpcFactory* to define which implementation of an individual component needs to be instantiated.

In the example provided below, the Device Descriptor requires the creation of Channel with type "HTTP" and Message Mappers of type "urlencoded" and "json".

The first section of the *Device Descriptor* after the header contains a declaration of all attributes exposed by the device.

For each attribute the device designer can specify the following properties:

ID Textual label used to univocally identify the attribute

Type Data type of the attribute

Permission The allowed operations on the attribute (read, write, ...)

```
<attributes>
        <attribute id="temperature" type="float" permission="read"/>
        <attribute id="room" type="int" access="static"
                value="5" permission="read"/>
        <attribute id="pressure" type="float" permission="read"/>
        <attribute id="period" type="integer" permission="write"/>
</attributes>
```

Listing 2: Attribute declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<device ...
  xmlns:ht="http://perla.dei.org/channel/http">
  ...
  <channels>
    <ht:channel id="http_ch_01">
    </ht:channel>
  </channels>
</device>
```

Listing 3: Channel declaration and configuration

Access Type of access. Can be set to DYNAMIC (the attribute is generated by the device) or STATIC (the attribute value is constant). Static attributes are required to declare their value

The Channel section defines which Channels have to be used to interact with the remote device. As in the example below, the set of configuration attributes required by a Channel depends entirely on the type of the Channel itself. The following code snippet also demonstrates how namespaces are employed in the Device Descriptor to link configuration parameters with a specific component type.

The I/O Request section allows PerLa user to define a series of native, primitive functions that can be performed on the remote device. These primitives will then be used in other sections of the descriptor to specify more complex behaviours. I/O Requests are described with the same paradigm previously seen in the *Channel* section; an XML namespace defines the type of request, which is then used to instantiate the right object using the given parameters.

```
<requests>
        <ht:request id="weather"
                host="http://api.openweathermap.org/data/2.5/weather"
                content-type="application/json"
                method="get" />
        <ser:request id="query_temperature" />
        <ser:request id="reset_device" />
</requests>
```

Listing 4: IOrequest configuration

Messages and *Message Mappers* are defined in the Message section. This part of the Device Descriptor is used both for defining the structure of all messages used to communicate with the device and for binding attributes to a specific message field. This section uses the familiar namespace paradigm for selecting a particular implementation of FPC component. In the example below, a Mapper for JSON objects is selected using the "json" namespace. It is important to note that, once a particular data type is selected, the user can define the structure of a message using concepts and idioms typical of that format.

The last Device Descriptor section describes how the primitive functions made available by the network node have to be combined to create a data record. Each operation declared in this section defines a Script, which is executed by the FPC whenever one of the emitted attributes is requested with a PerLa query.

Additional information regarding FPC Scripts can be found in the Script Engine section of this document.

3.2 Channel

A *Channel* is a software unit capable of performing I/O operations. It defines a network-independent interface that can be used by high-level components of the PerLa Middleware, regardless of the particular protocol stack employed. Different concrete channel implementations are available in the PerLa Middleware; each channel is committed to the management of a single communication technology.

A channel exposes the following APIs:

```
<messages>
        <js:message id="weather_msg">
                <js:object name="msg" type="object">
                        <js:object name="main" type="object">
                        <js:object name="temp"
                                type="float"
                                qualifier="attribute"
                                attribute-id="temperature" />
                        <js:object name="pressure"
                                type="float"
                                qualifier="attribute"
                                attribute-id="pressure" />
                        </js:object>
                </js:object>
        </js:message>
</messages>
```

Listing 5: Message declaration – 1

```
<operations>
        <get id="get-weather">
                <i:submit request="weather" channel="http_ch_01"
                        variable="result" message="weather_msg"/>
                <i:put variable="result" attribute="temperature"/>
                <i:put variable="result" attribute="pressure"/>
                <i:emit/>
        </get>
</operations>
```

Listing 6: Message declaration – 2

IOTask submit(IORequest, IOHandler) Submits an *IORequest* for execution, and specifies the callback handler to be invoked when the result is ready

setAsyncIOHandler(IOHandler) Sets a callback handler for managing asynchronous communications initiated by the remote device

Channel operations are asynchronous by nature; users send requests by means of the submit method, and receive results through the specified *IOHandler*. Request execution may be monitored or cancelled using the *IOTask* object returned upon submitting a new *IORequest*. Synchronous data transfers can be tracked despite the inherently asynchronous operating principle of the Channel; in case of synchronous communication the association between the request that triggered an I/O operation and the result of such operation is maintained by the handler callback.

As can be seen by examining the methods in the *IOHandler* interface:

complete(IORequest, Payload) Asynchronously invoked when the I/O request completes successfully

error(IORequest, error) Asynchronously invoked when the I/O request completes with an error

The Channel component is also designed to handle asynchronous data transfer operations initiated by the remote device, a scenario that may occur whenever the node independently streams a series of data records to the PerLa system. Payload data corresponding to such communications is made available through a single IOHandler, usually associated with the FPC's *Channel Manager*.

A Channel usually implements a complete communication protocol that allows the FPC to effectively and efficiently communicate with the remote device or service. The use of multiple high-level, moderately specialised Channels is preferred over a limited set of generic, multipurpose Channels, since the latter design would put too much overhead on both the FPC and the final user.

Ideally, each PerLa Channel should implement a protocol corresponding to the Application layer of the OSI model. This is in direct contrast with the previous middleware design, which fostered the creation of a small set of re-useable Channels that only managed *Transport layer* communications. Moving high-level protocol

management from the FPC to the Channel is beneficial in that it promotes the use of readily available third-party communication libraries. As an example, a Channel implementation of the FTP protocol could use one of the immediately available FTP Java libraries. This would not be feasible if application and transport logic were to be split between the FPC and the Channel. The creation of new *Channel* objects at runtime is performed using a *ChannelFactory*. The PerLa Middleware contains a specialised ChannelFactory for each Channel implementation (i.e., the *HttpChannelFactory* only creates instances of the *HttpChannel*).

3.3 Script Engine

The Script Engine is a newly introduced component in the PerLa middleware, specifically designed as a tool for manipulating data received or sent through the *Channel*. Its purpose is to act as an "*impedance matcher*" between the record-based world of PerLa and whichever data structure is in use on the remote sensing network. PerLa Scripts are composed of a series of simple imperative instructions, whose main functions are:

- Create and customise I/O requests with information that dynamically changes at runtime
- Submit I/O requests to a remote device
- Convert data received from the remote devices into one or more records

PerLa Scripts employ all primitive entities described in the previous sections of this document (channels, I/O requests, mappers, etc.), and connect them to implement complex behaviours.

3.4 FPC Factory

New FPC objects are instantiated at runtime by the FPC Factory. The starting point for the creation of an FPC is the Device Descriptor, an XML document which contains a machine parseable description of a single sensing device. Device Descriptors are organised in different sections, each of which defines the configuration of one of the aforementioned FPC modules. The Fpc Factory can receive new

Device Descriptors directly from the node being connected (Plug&Play behaviour), or from another entity that acts on behalf of it (off-band behaviour). The latter approach allows devices which are not capable of autonomously transmit their Device Descriptor to be registered on the PerLa framework.

A reference to each FPC is stored in the FPC Registry, a component of the HAL Layer that is responsible for maintaining a complete index of all devices accessible through the PerLa framework. Thanks to the Registry, PerLa user can discover sensing nodes through capability-based queries, and retrieve the FPC objects that can be used to interact with them.

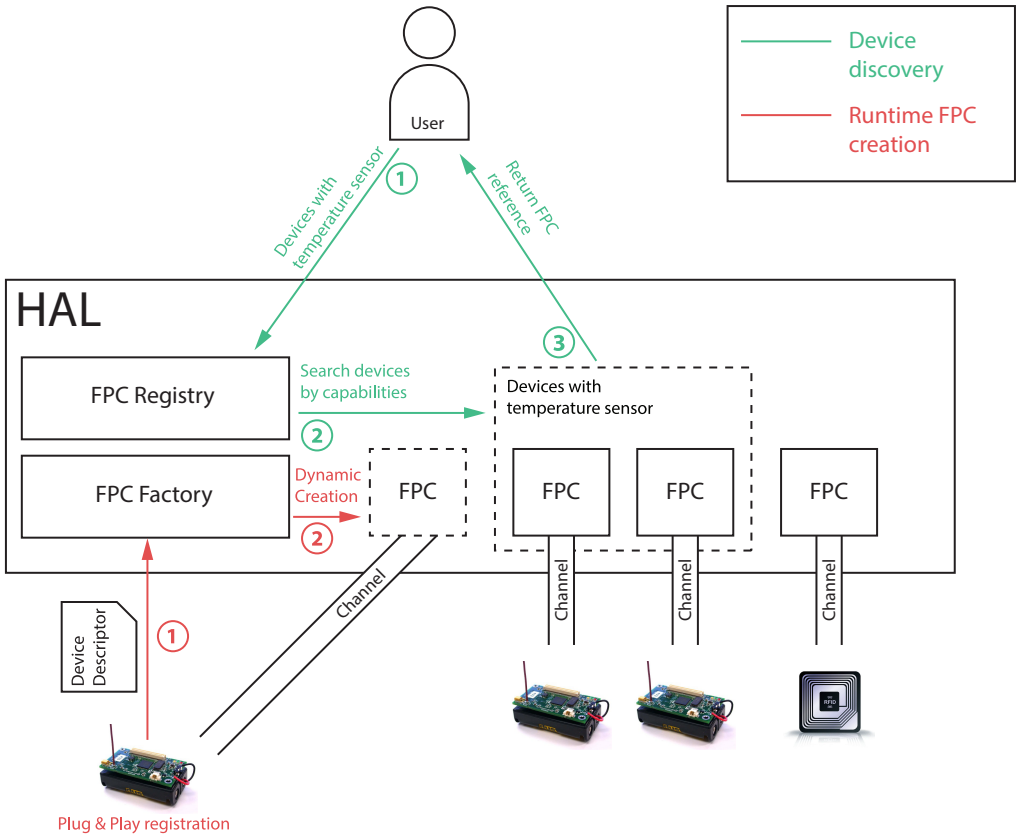


Figure 3: Dynamic FPC creation process

4 HTTP protocol integration in PerLa

In its first versions PerLa connects sensors through TCP/IP protocol, while nowadays some information could be retrieved from other systems (e.g. web services) and not only from sensors.

There exists plenty of Web Services offering APIs (JSON/REST or SOAP), like weather forecast services (providing temperature, pressure and other weather-related parameters). These APIs give information that PerLa could model like a virtual sensor.

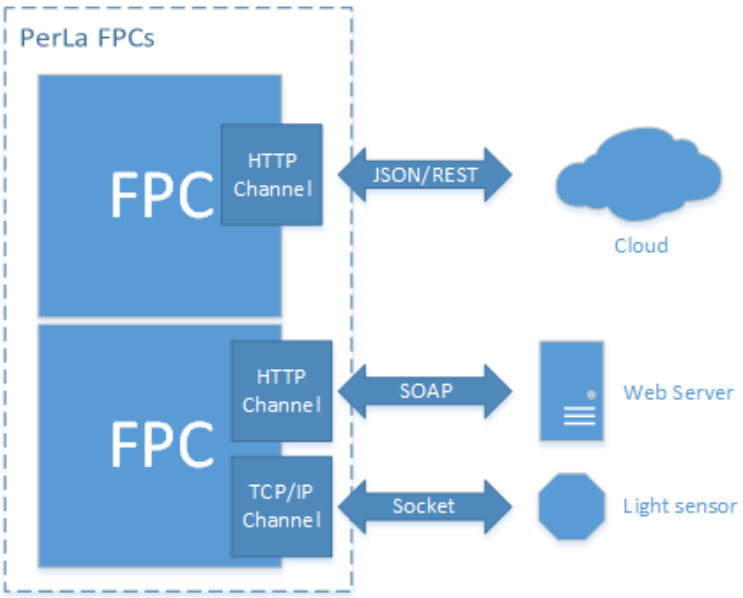


Figure 4: FPC Channels

The enabling feature for this new paradigm is the implementation of a *Channel* supporting the HTTP communication protocol. The real challenge is to create this channel as generic as possible, so it will be independent from the content-type used by the HTTP call for transferring data (in the *request* or in the *response*).

4.1 PerLa Channel design

As presented in 3.2 the PerLa *Channel* is the component responsible for the communication between FPC and real device. In this context the device is represented by a common Web Service with an HTTP interface (ex. SOAP or JSON/REST).

In order to define and configure a channel PerLa offers three base classes:

AbstractChannel is the abstraction of any communication channel between PerLa (FPC in particular) and the real device. We can consider channels based on TCP Socket technology, WebSocket, for advanced device, or HTTP protocol.

This object takes care of sending a generic *Request*, creating a *Response* and adding it to the queue passed to the FPC.

ChannelRequestBuilder is the object used by the FPC for creating the requests to be passed to the channel. Each request class has its *ChannelRequestBuilder*.

Request is the object created at run-time when a FPC needs to retrieve some data from a real device. After being created by *ChannelRequestBuilder*, it must be configured with some parameters. For example a JSON/REST service may need to receive the name of the city for retrieving a particular temperature.

The complete process of how the channel handles a request is shown in Figure 5.

Usually the channel provides synchronous requests; those request are added to the queue shared between the channel and the FPC. The FPC can then use immediately the response (synchronous call) or it can postpone this action and retrieve it just when it is needed (asynchronous call).

PerLa *Channel*s are build dynamically at run-time through the translation of a Descriptor (Java representation of XML descriptor) into PerLa objects.

This translation is based on the following Java objects:

ChannelFactory is responsible of XML channel descriptor validation. So it is used by a FPC for creating Channel during the PerLa start-up phase

ChannelRequestBuilderFactory is responsible of the validation of the XML

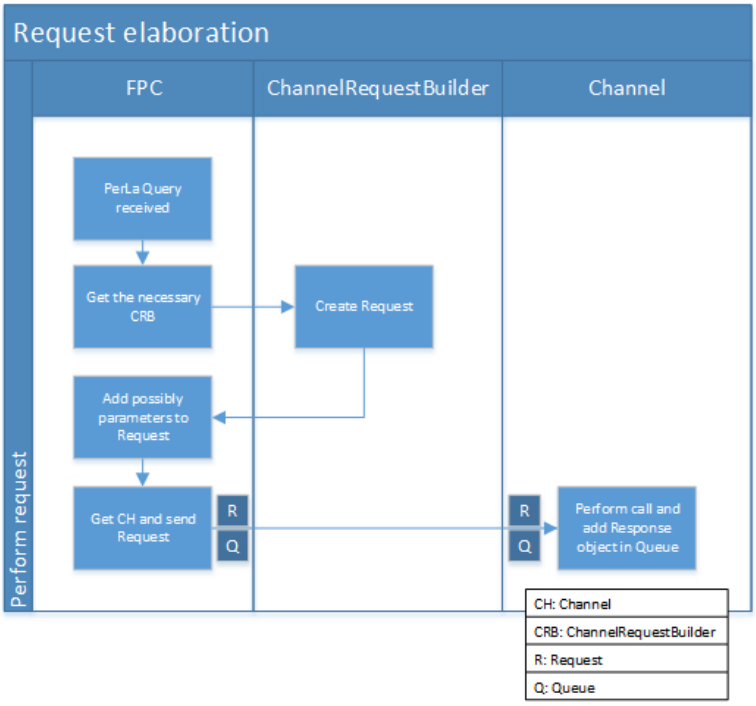


Figure 5: Request elaboration

request descriptor. So it is used by FPC for creating a *ChannelRequestBuilder* during the start-up

ChannelDescriptor is the Java representation of *channel* XML tag

RequestDescriptor is the Java representation of *request* XML tag

4.1.1 HTTP Channel implementation

HTTP Channel implementation has been realized starting from the classes described before.

All XML tags of HTTP context are refereed to *name-space http://perla.dei.org/channel/http*.

HttpChannel is the object responsible of performing HTTP call, using *HttpChannelRequest* and returning a *ChannelResponse*

HttpChannelRequestBuilder generates the *HttpChannelRequest*

HttpChannelRequest wraps the HTTP request with completed URL (host, path and query), HTTP method and eventually entity and content-type

HttpChannelFactory validates the channel descriptor and creates the *HttpChannel* object

HttpChannelRequestBuilderFactory parses the XML file, wrapped in *HttpRequestDescriptor*, and creates an instance of *HttpChannelRequestBuilder* for the related FPC

HttpChannelDescriptor is a channel instance responsible of the *HttpChannelRequest*s. It is a Java object that translates the *channel* tag having like attributes just an *id*

HttpRequestDescriptor is the Java class mapping HTTP *request* XML tag inserted in the device descriptor.

HttpChannel Is created by the *HttpChannelRequestBuilderFactory* at start-up, and associated to the correspondent FPC. It is then invoked by the FPC using the *submit* method, which, after having consumed a *HttpChannelRequest*, returns a *ChannelOperation* containing the logic with call result. This method is inherited by the *AbstractChannel* object, that implements the *Channel* interface.

The *submit* operation is performed by means of the *handleRequest* method, that dispatches the GET, POST, PUT and DELETE *HttpChannelRequest*, respectively, to *handleGetRequest*, *handlePostRequest*, *handlePutRequest* and *handleDeleteRequest*. All these methods are implemented following the REST architecture.

For the implementation proposed in this work the Apache HTTP Component library[1] has been used.

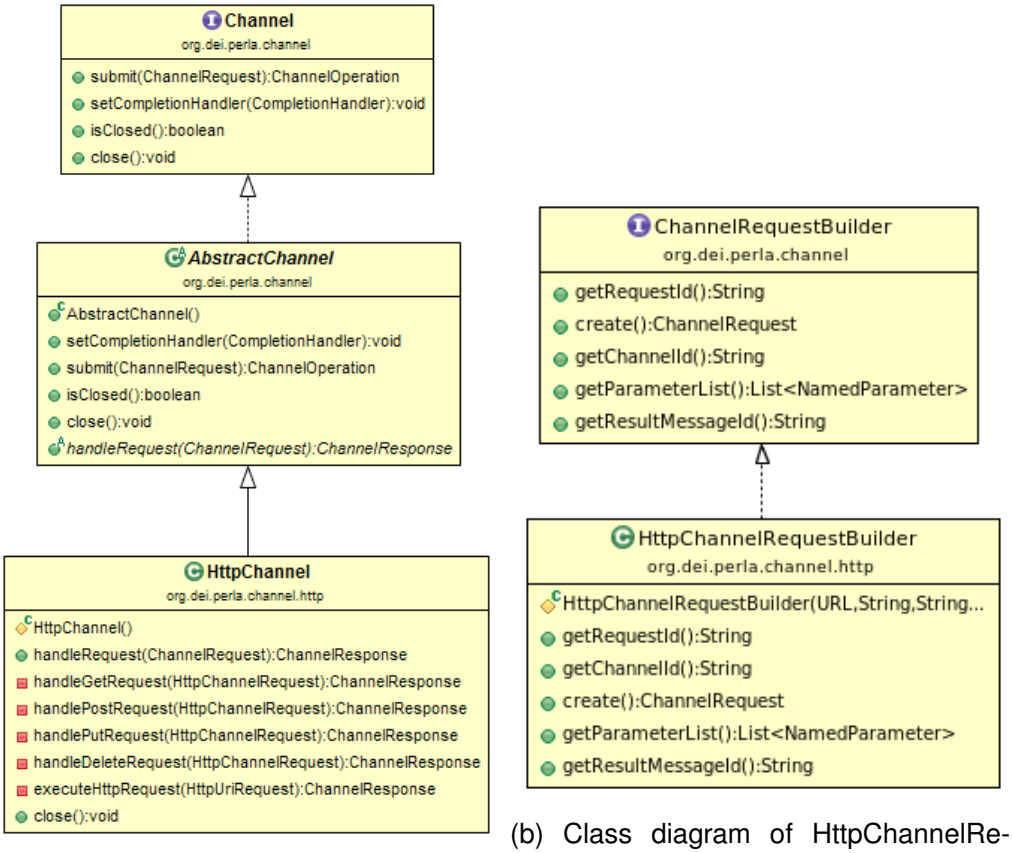
The object *CloseableHttpClient*, created into *HttpChannel* constructor, works as transporter of *HttpUriRequest*, built by PerLa Channel in *handle* methods.

The class diagram of the *HttpChannel* related classes is shown in Fig. 6a.

The class diagram is reported in Figure 6b.

HttpChannelRequest *HttpChannelRequest* models the content to be sent through the channel, so has essentially four parameters: *(i)* HTTP method, *(ii)* URI,

[1] `https://hc.apache.org/`



(a) Class diagram of HttpChannel

(b) Class diagram of HttpChannelRequestBuilder

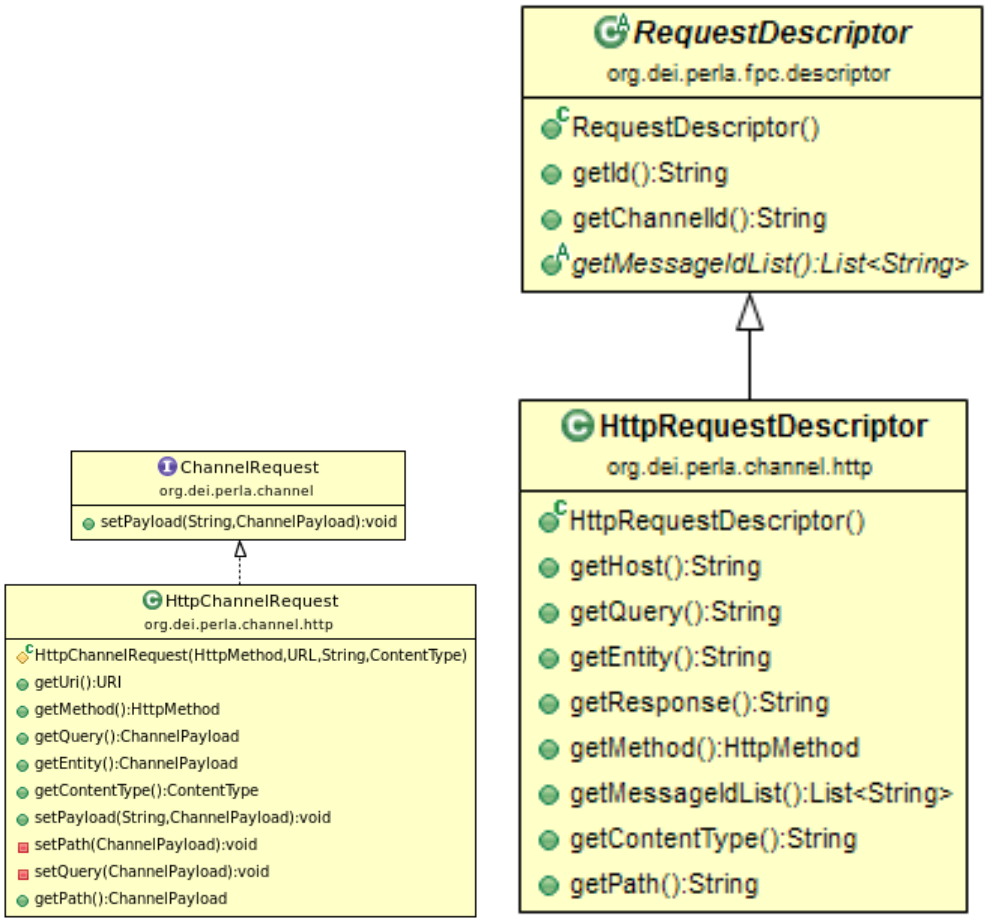
Figure 6: HttpChannel class diagram

(iii) Content-type and *(iv)* Entity

The *Entity* parameter is dynamically set by the FPC using the *setParameter* method.

HttpChannelRequest implements all methods of *ChannelRequest* interface, which is used by the FPC on a higher abstraction level (Fig. 7a).

HttpChannelFactory This class is responsible of the *HttpChannel* creation starting from the *HttpChannelDescriptor*. It validates the descriptor and initializes the channel applying the attributes (e.g. *id*, channel identifier).



(a) Class diagram of HttpChannelRequest

(b) Class diagram of HttpRequestDescriptor

Figure 7: HttpChannel class diagram (continued)

HttpChannelRequestBuilderFactory It is one of the critical point of HTTP integration because it must validate the XML *request* tag and check its consistency with the REST architecture. Attributes *id*, *channel-id* and *host* are always mandatory while attributes *path* and *query* are always allowed. All controls are performed into the *create* method, which throws an *InvalidDeviceDescriptorException* when a check fails.

HttpChannelDescriptor *HttpChannelDescriptor* is the POJO (*Plain Old Java Object*, an ordinary Java object) that translates the XML *channel* tag. The channel

identifier is used for associating each request with its channel. An example of the *Channel* tag is provided below:

```
<?xml version="1.0" encoding="UTF-8"?>
<device ...
  xmlns:ht="http://perla.dei.org/channel/http">
  ...
  <channels>
    <ht:channel id="http_ch_01">
    </ht:channel>
  </channels>
</device>
```

Listing 7: HTTP Channel XML descriptor – 1

HttpRequestDescriptor Request descriptor is a POJO that maps all the *request* related tags. For correclty understanding the mapping it is necessary to know how the XML *request* tag is composed. All *request* tags are characterized by nine attributes:

id is a required string identifier of request.

channel-id is a required string identifier of channel sending this request.

host is the required string (URL) identifying the host to which the HTTP request has to be sent.

path is the optional identifier of tag *message* that represent a dynamic path.

query is the optional identifier of tag *message* that represent a dynamic query.

entity is the identifier of tag *message* that represent the entity (content) of HTTP request and it is required for POST and PUT requests

method is an optional enumeration value that specifics HTTP method for the request (get, post, put or delete); the default value is *get*

content-type is the optional content-type specified in HTTP request. The default value is */* (Known as *wildcard* content-type)

response is the identifier of the tag *message* that represent the response content of HTTP request. It is required for post and get request

The following script represent a POST request example:

```
<?xml version="1.0" encoding="UTF-8"?>
<device ...
  xmlns:ht="http://perla.dei.org/channel/http">
  ...
  <requests>
    <ht:request id="post_req"
      channel-id="http_ch_01"
      method="post"
      host="http://mysite.com"
      path="req_path_id"
      query="req_query_id"
      response="req_response_id"
      content-type="application/x-www-form-urlencoded"
      entity="req_entity_id"/>
  </requests>
</device>
```

Listing 8: HTTP Channel XML descriptor – 2

Looking at the class diagram in Figure 7b it is possible to understand the relationship between methods and attribute of the *request* tag.

HttpRequestDescriptor implements the *getMessageIdList*, an abstract method inherited by *RequestDescriptor*, usable by FPC to parse the request.

5 PerLa on TinyOS-based Motes

A TinyOS application has a particular structure composed by some files that are then compiled and installed on motes. There are two types of components in *nesC*[2]: *modules* and *configurations*. *Modules* provide the implementations of one or more interfaces (collection of commands and events). *Configurations* are used to assemble other components together, connecting the different interfaces used by

[2] http://nescc.sourceforge.net/

components. It is important to remind that TinyOS is an operating system with an event-driven architecture.

The PerLa nesC implementation is split in 4 main files:

PerLaSensorAppC.nc This file contains the top-level configuration: it wires together the component that are used in the application

PerLaSensorC.nc This file contains the implementation of commands/ events provided/used by the application. It is the biggest part of the code, that specifies the behaviour that motes will assume in different situations (at boot, when a timer fires, when they receive a message and so on)

PerLaMessage.h This file specifies the structure that a message must assume in the PerLa system

PerLaSetting.h This file contains some structures, used in the application, that the producer can modify as it prefers. This will allow him to avoid a deep intervention on the code of the application by accessing only this file.

5.0.2 PerLa Message Structure

A *TinyOS* (TOS) message is mainly composed by three parts: *header*, *data* and *metadata*. *Header* contains general information about the message for the communication; *data* contains the values that have to be exchanged; *metadata* contains additional communication information. The message structure is shown in Fig. 8:

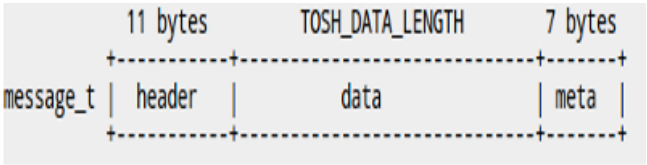


Figure 8: Standard TinyOS message structure

Since changing the TOS header is not recommended we chose to create and encapsulate a customized PerLa message within a standard TOS message. By doing this, total control over the PerLa message is retained.

The header of a PerLa Message, whose declaration is shown in listing 9, is composed of the following parts: *Id* is the identifier of the mote that sends the message,

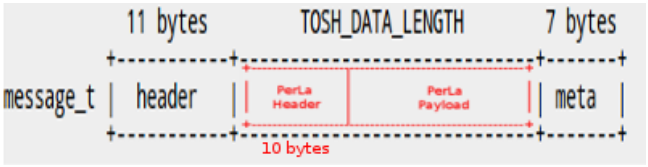


Figure 9: PerLa message structure

```
typedef nx_struct perlaHeader {
    nx_uint16_t id;
    nx_uint16_t timestamp;
    nx_uint16_t type;
    nx_uint16_t numPckt;
    nx_uint16_t numPcktToReceive = 0;
} perlaHeader_t;
```

Listing 9: The Header of a PerLa Message

timestamp contains the date and hour in which the message was composed and sent, *type* specifies the kind of the message ("0 Device Descriptor, 1 periodic sampling (+ one-shot), 2 event based sampling, 3 end of periodic sampling, 4 end of event based sampling, 5 periodic response, 6 event based response").

numPckt and *numPcktToReceive* are two fields used when the amount of information to be sent exceeds the maximum size of `message_t`, and data must be splitted over more packets. Every time that the PerLa payload is split, the PerLa header is replicated with different values of this field: the first one is used to keep the received packet ordered allowing to rebuild the original data, the second one tells to the receiver how many packets it must receive before the communication of the information ends.

The PerLa payload, instead, has the following structure:

```
 typedef nx_struct perlaPayload { nx_uint8_t
data[TOSH_DATA_LENGTH-sizeof(perlaHeader)]; } perlaPayload_t;
```

The payload is a simple array of bytes. There is no need to specify a more complex structure, since the underlying idea is to keep it as generic as possible.

5.0.3 Design Details

In order to guarantee an easier migration from TinyOS to some other embedded operating systems, and to improve code maintainability, some functions and a task that separate the logic from the applicative code have been implemented:

splitMessage() it is called when a message is too long to be sent in a single packet. It takes the complete message, splits it into chuncks and replicates the header for every partial message. Moreover, it fills the field of the header in which it is specified with the total number of packets in the sequence and the number of each packet in the sequence itself. When a PerlaMessage is composed, this function puts it into a transmission queue and starts the task used for sending packets

sendMessageInQueue() this is the task responsible for sending the content of the transmission queue. It checks if the queue is not empty and sends the first packet. Once the packet is sent, it calls itself recursively until the queue has been emptied.

setHeader(uint16_t type) it takes as an input the type of the message (specified by the protocol) and fills the id, timestamp and type fields of the header of a perlaMessage.

composeAndSend() takes the header and the payload with the field already compiled, puts them into a perlaMessage and sends it. If the size of the perlaMessage is too big with respect to the size of a TinyOS message, it calls the splitMessage() function.

6 Cassandra integration

The output of a PerLa query can be easily dumped in a Cassandra Database using the `DatabaseWrapper` class available in the perla-cassandra module. Apache Cassandra is a hybrid key-value and column-oriented open source distributed database management system designed to handle large amounts of data across many commodity servers. Unlike traditional RDBMS, rows in a Cassandra table don't necessarily share the same set of columns, and a column may be added to one or

multiple rows at any time.

6.1 The DatabaseWrapper class

The `DatabaseWrapper` is responsible for saving new data records to a Cassandra database. As suggested by its name, this class is a wrapper that contains a record producing object. Its interface is composed of 3 methods, which correspond to the main data access primitives available in an FPC. Every record produced by an FPC enclosed in a `DatabaseWrapper` is automatically dumped into a Cassandra database. The name of the output table can be retrieved from the `DatabaseTask` object returned when starting a new query.

As can be seen in figure 10, the `DatabaseWrapper` class uses a custom `Handler` to save data inside a Cassandra table. The records produced by the FPC can also be relayed to an additional user-specified handler, thus allowing additional components to be daisy-chained for real-time data analysis.

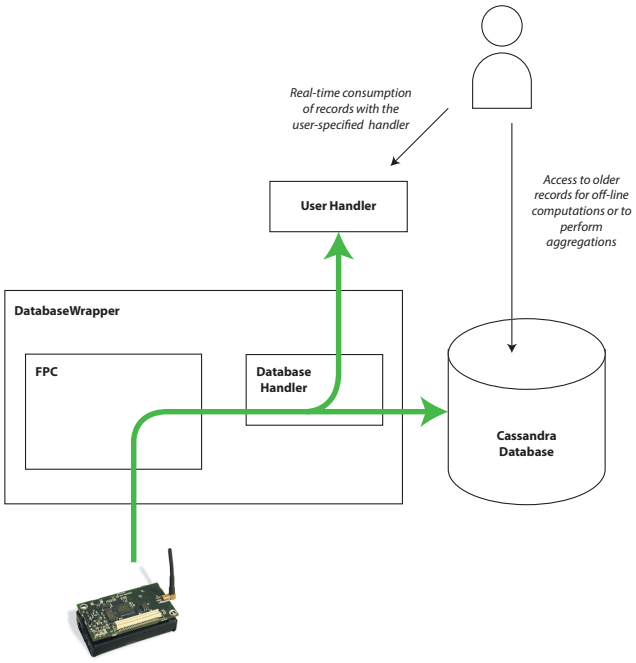


Figure 10: DatabaseWrapper architecture

Records are inserted inside the database by the `DatabaseHandler`. This object

is designed to gracefully support interruptions in the record flow, and to append data in the correct table even when a query is paused and resumed. Every record is complemented with a randomly chosen Identifier prior to its insertion in the database.

7 Ongoing work

7.1 Integration with SensorML

SensorML is a standard for the description of sensing devices developed by the Open Geospatial Consortium. In its first version the standard was composed only by a single specification, whereas the 2.0 version of the standard is composed by *SensorML* and by *Sensor Web Enablement Common Data Model*, that was included in the first version of the SensorML standard and has been extended to support the other Sensor Web Enablement standards developed by OGC. Supporting *SensorML* would allow all devices already described using this standard to be immediately compatible with the PerLa Middleware.

The integration with *SensorML* is an ongoing project aimed at expanding the compatibility of PerLa. The proposed design recommends the introduction of a new mapping parser for translating *SensorML* node definitions into a PerLa-compatible *Device Descriptors*. The advantage of such solution lies in the possibility of reusing all existing PerLa components.

7.2 Distributed PerLa

PerLa currently acts as centralized server architecture. In case of small sensor networks this will not be a problem, since the central server has the capability to handle the whole network traffic and react to the appropriate queries. However since in complex and wide sensor networks problems may arise due to the heavy network load, distribution may result as a crucial aspect for the PerLa system.

Distribution is possible since in these networks some nodes with higher capabilities, both for computation and routing — usually defined as *gateways* – are

located somewhere in the network path.

In addition gateways are the ideal candidates for the computation load distribution since they usually *i)* gather data from several nodes and *ii)* have much more computing power than needed for routing purposes.

On gateways a "light" version of the PerLa server can be installed and executed. The central server can then decide, depending on the query, how to use gateway. In some queries all the data coming from the underlying sensor may be necessary at central level, while in some cases an aggregate value (e.g. the average temperature detected by sensors located in a given zone, connected to the same gateway) may be enough for the central server needs.

In the second case the PerLa central server just need to know that the "*complex sensor*" (the gateway) outputs a stream in a given format (and the format is described as it were a standard sensor using the FPC XML descriptor), aggregating data coming from a part of the sensor network, no matter which are these sensors (that on the other hand must be well known to the PerLa gateway level) that become absolutely and totally transparent to the central system.

However, before the reenginering process, it was really hard to understand if such "light" version of PerLa can be obtained extracting only the necessary modules from the original implementation.

Now it is possible to produce many different "light" version of the PerLa server. The main modules involved are the *Channel* module and the two query executor (*OnTheFlyQueryExecutor* and *StandardQueryExecutor*) modules.

The output of the subsystem can be seen from the central server as a sensor (e.g. like a Web Service sensor) that provides as fields the ones of the query that runs on it.

In such a way the subsystem can be described by an XML descriptor in the central server, and its data could be used for computing other HLQs. From a database point-of-view, the subsystem output results are simply a view on data coming through a specific gateway.

7.3 Context-awareness

Context-awareness is a property inherent to an autonomic Pervasive System and requires a clear definition of what context is and how the context parameter values can be extracted from the real world.

A possible definition of context is the following: "any information that can be used to characterize the situation of an entity". Pervasive Systems are aggregations of different and multiple devices, which are spatially distributed and possibly mobile: these devices allow monitoring different kinds of physical phenomena for application support. However, some of these data can also be used to detect the situation of the entities present in the environment, in order to characterize the context and to provide the actual values for the context parameters.

Most of the existing solutions integrate both the operational layer and the content management layer in an unique one. Differently from these approaches, we separate them in two different layers, while embedding in the same language the functionalities of both: context is analyzed in terms of observable entities, which have some symbolic representation within the system and some of which correspond to numerical values gathered from the environment sensors. Examples of these values are the time of day or the GPS coordinates and we call them numerical observables. Numerical observables are gathered from the environment by means of PerLa.

Gathering context data from the environment requires a simple interface, possibly based on a declarative approach, which, on one side, interacts with the network of highly heterogeneous physical devices and, on the other, is correctly interfaced with the internal, symbolic representation of context, based on a context model.

The fact that PerLa provides a declarative language allows a very easy integration between the activity of querying numeric observables and the activity of querying other observables like the role of the system user or the current step of the workflow process.

The following sections contain an overview on two active projects aiming at integrating the PerLa system with context management features. The first one proposes a context management language based on the Context Dimension Tree

model, named PerLa Context Language (PerLaCL), whereas the second intends to integrate PerLa with a set of Context Oriented Programming extension (Schreiber et al. [2012b, 2011]).

7.3.1 PerLa Context Language

According to the Context Dimension Tree (CDT) model the set of possible contexts of the environment can be modeled as a labeled tree composed of dimensions and concepts nodes. The former are used to capture the different characteristics of the environment, while the latter are used to represent the admissible values that can be assumed by the dimensions. Both dimensions and concepts can be semantically enriched using attributes, that are parameters whose values are provided at runime. Moreover, the "tree" term suggests that the designer can model the environment using the preferred granularity, nesting more than one level of dimensions with the unique restriction that every dimension can only have concept children and vice versa. This constraint imposes that the node colors alternate while descending the tree, as clearly shown in figure 11a, where the visual representation of CDT of our running example is shown, containing the fundamental aspects of context in a vineyard scenario. For example the dimension Role captures the actors involved in the wine production while the Phase dimension, together with the Risk dimension, model the various phases and the possible risks that could compromise the final product.

In order to denote that a dimension has assumed a certain value we use the ¡Dimension = Value¿ notation, called a context element. A context can then be formalized as the conjunction of one or more context elements.

As an example, in the CDT shown on the left side of figure 11, a possible context could be the one graphically represented in figure 11a as a subtree of the one of the original one. It is worth noticing that not all possible subtrees are valid contexts. This is the case of Figure 11b where the dimension Role assumes the values Driver and Farmer simultaneously (the children values of one dimension are always to be instantiated in mutual exclusion). The designer can specify further constraints forbidding some context elements to be used in the same context definition. These

constraints, called useless context constraints are depicted using a line that links the mutually exclusive values. In our example, the truck Driver will never be involved in any of the activities which are pertinent of the grapes Growth phase.

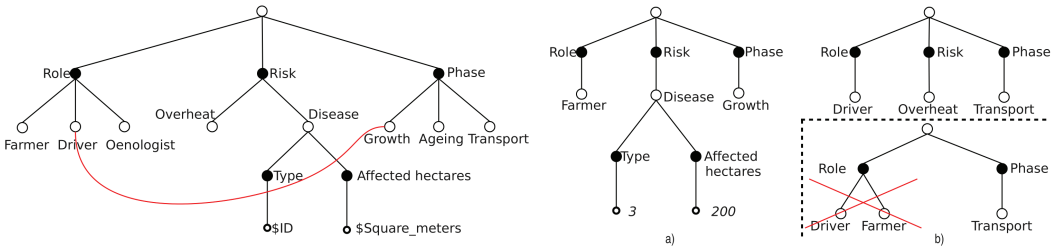


Figure 11: A Context Dimension Tree

The syntax of PerlaCL has been divided into two parts, called CDT Declaration and Context creation.

CDT Declaration This part allows the user to specify the CDT, i.e. all the application-relevant dimensions and values they can assume. As an example of its usage we report the syntax to define completely the Risk dimension of our running example as well as the definition of the Driver-Growth useless context constraint.

A set of CREATE DIMENSION CONCEPT statements allows to declare the dimensions as well as their concepts nodes. The sibling of an internal dimension can be specified adopting the CHILD OF clause, otherwise the dimension is meant to be a child of the tree root. When creating a concept of a dimension, the designer must specify the name and the condition for assuming the specified values by means of numeric observables that can be measured from the environment (WHEN clause). When, instead, the design requires the presence of attributes the CREATE ATTRIBUTE clause must be used, using the $ sign as a prefix before the name of the attribute, meaning that its value will be supplied by the application at runtime. The EXCLUDES clause is employed to express useless contexts constraints.

Context declaration This part of the syntax allows the designer to declare a context on a defined CDT and control its activation by defining a contextual block, which is composed by four fundamental parts, called components:

```
1
2  CREATE DIMENSION Risk
3      CREATE CONCEPT Disease
4          WHEN getInterestTopic()=‘‘disease’’
5      CREATE CONCEPT Overheat
6          WHEN temperature > 30 AND brightness > 0.75;
7      CREATE DIMENSION Type
8          CHILD OF Disease
9          CREATE ATTRIBUTE \$id
10
11 CREATE DIMENSION AffectedHectares
12     CHILD OF Disease
13     CREATE ATTRIBUTE $squareMeters
14
15 CREATE DIMENSION Role
16     CREATE CONCEPT Driver
17     EXCLUDES Phase.Growth
```

Listing 10: CDT declaration

ACTIVATION component allows the designer to declare a context, using a CREATE CONTEXT clause and associating a name to it;

ENABLE component introduced by an ON ENABLE clause, allows to express the actions that must be performed when a context is recognized as active;

DISABLE component introduced by an ON DISABLE clause is the counterpart of the previous one, allowing to chose the actions to be performed when the declared context stops being active;

REFRESH component instructs the middleware on how often the necessary controls must be performed.

7.3.2 Integrating PerLa in Context-Oriented programming languages

PerLa with its queries allows to monitor the entire life cycle of the information.

Moreover, PerLa has been designed to work with several data streams, produced by the sensing devices; the Low Level Queries allow to set their working mode: the sampling intervals, which data must be selected and the computation to perform on the sampled data. With PerLa CL (see Sect. 7.3.1), the designer

```
1
2  CREATE CONTEXT Growth Monitoring
3  ACTIVE IF phase = ‘‘growth’’ AND role = ‘‘farmer’’
4         AND Disease.Type=3
5         AND Disease.Affected_Hectares = 200
6
7  ON ENABLE (Growth Monitoring):
8      SELECT humidity, temperature
9      WHERE humidity > 0 AND temperature>0 SAMPLING EVERY 6 h
10     EXECUTE IF deviceLocation = ‘‘wineyard’’
11
12 ON DISABLE:
13     DROP Growth Monitoring
14
15 REFRESH EVERY 1d
```

Listing 11: Context declaration

can define a sort of *contextual dynamic view* on a data stream. On the contrary, COP is not directly aware of how information is provided; in fact it is not directly responsible of sensors, but it uses the information provided by them, to perform behavioral variations. Adopting COP, the developers must implement a mechanism to monitor continuous data sources, in order to provide contextual information.

Moreover, PerLa is very suitable for context distribution; the CDT model deals naturally with contexts belonging to different groups of sensors and to distributed instances of the application. The nodes at the lower level of the tree could be used to abstract several instances of a dimension, creating a *local* CDT for each instance. In this way, context data can be distributed to different locations, leading to the introduction of a *combined* CDT comprising a primary CDT and one or more *local* CDTs. An example of CDT distribution is provided by the Green Move application, in which – for privacy – a portion of the CDT is maintained locally to the user's devices and is used to complete the context-based data filtering Panigati et al. [2012].

Also COP is oriented to application distribution; in fact, several threads may exist for each local instance and they adapt their behavior differently. However,

with COP it is only possible to implement the behavioral variations of the instances; mechanisms to compose information coming from different local contexts, in order to infer a higher level context, and for data sources monitoring are not supported. For this reason, it may become necessary to introduce dedicated components to generate significant contextual information, starting from rough data provided by sensors, and to decide which layer must be activated on the application.

The PerLa middleware has the components already designed for this purpose: the CM for what concerns the context and the application actions management, and other low level components for what regards the data sources monitoring, their operational impact being hidden to the user. Designers, through the PerLa CL, are relieved of the responsibility to develop dedicated components for context management which could result a rather complex task. With the possibility to define new contexts by composing other contexts at run-time, it becomes easier to specify the desired number of contextualized actions.

Enacting behavioral variations

The semantics of the *ON ENABLE* and *ON DISABLE* clauses of PerLa CL could apparently look similar to that of the ***with*** and ***without*** statements of COP. Considering the *ON ENABLE* clause, it sets which data must be provided, and then it utilizes these data to change some parameters, according to the corresponding concept in the CDT.

If the context is active and therefore the condition in the *WHEN* clause of the corresponding concept is satisfied, an established action is performed.

The following PerLa code (Listing 12) shows that, by installing a new air outlet in the offices, its activation becomes possible in case of persistent smoke. This new function is available at the same time as we plug the air outlet into the system, without the necessity of further changes to it. In addition, also a new configurable air conditioner has been installed in the offices. It can work in different modalities, depending on different properties of the room environment (it can cool down or heat up the temperature and it can dry off the environment). The corresponding CDT is shown in Figure 12:

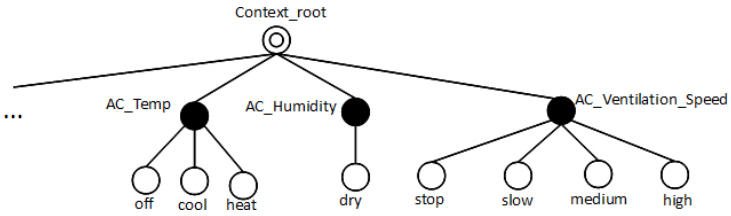


Figure 12: Context Dimension Tree (CDT) – Addition

```
1
2  CREATE CONTEXT SmokeMonitoring
3  ACTIVE IF Location = 'office'
4      AND Smoke = 'persistent'
5  REFRESH EVERY 1h
6
7  ON ENABLE (SmokeMonitoring) :
8      SELECT smoke
9      SAMPLING EVERY 10 m
10     EXECUTE IF EXISTS (smoke)
11     SET PARAMETER air_outlet = TRUE
12     SET PARAMETER alarm = TRUE
13     SET PARAMETER speed = 0
14
15 ON DISABLE (SmokeMonitoring) :
16     DROP SmokeMonitoring
17     SET PARAMETER air_outlet = FALSE
18     SET PARAMETER alarm = FALSE
```

Listing 12: The SmokeMonitoring example in PerLa

To implement the office example with COP, an external mechanism to monitor changes in context must be specified in addition to the definition of layers and when they have to be activated.

The proposed COP code (Listing 13) translates the PerLa example of Listing 12: the main thread *SmokeMonitoring* sleeps for ten minutes (cf. the PerLa *SAMPLING EVERY 10m* clause) and, when resumed, it controls the smoke level in the room; if it detects persistent smoke (cf. the PerLa *ACTIVE IF* clause), it activates the context *smokeRisk()* in which the ***with*** statement will activate the *SmokeLayer* in the class *ActiveActuators*, so executing the same actions that the

```
    when(SmokeMonitoring.smokeRisk()) { with(SmokeLayer); } }

class ActiveActuators { public activeAlarm() { // When this method is called
     by thread FireMonitoring, // the fire alarm is activated }

    ...

    layer SmokeLayer { activeAlarm() { // If the layer is activated, the air
    outlet will be opened Outlet.sendOpenCmd(); } }

}
```

Listing 13: The SmokeMonitoring example in COP

PerLa code invokes in the *ON ENABLE* clause.

This example shows that context in COP is more driven by events than by rough data; it also introduces other similarities between PerLa and COP, at least from a conceptual point of view. The concepts of partial components and partial methods could be considered based on the same idea of composing different entities to provide a new behavior; the PerLa context query and the COP context structure contain dedicated statements to declare when context changes and which actions must be executed in response.

Since PerLa can be adopted for applications performance monitoring, sensors can be configured for this purpose and the collected data can be used to change the working parameters of the monitored application, in order to increase its efficiency. For example, if the application response time must be decreased in case of heavy workload, at the risk of increasing the energy consumption, the proper dimensions and concepts must be introduced in the CDT and a simple CL query to the PerLa middleware will satisfy the request.

Therefore, the application developers must declare the significant context changes and which actions must be performed in different situations, with no need to implement anything at the operational level.

An extended architecture The most intuitive solution to the implementation of a Context-Aware self-adapting system should be to delegate everything related to the configuration of sensors, the intermediate computations on data, and contexts declaration and management to PerLa, and to use COP to perform

behavioral variations, using the contextual information provided by PerLa.

Considering that PerLa creates several data streams which include the values retrieved by sensors, a context can be viewed as particular **context data stream**, i.e. a normal PerLa stream, properly adjusted to provide only data related to the defined context. This operation can be performed by the CM: it manages the CDT and the actual contexts, and creates contextual streams accordingly. Some simple actions, such as those of the office examples, can be still performed by the CM.

COP, in turn, can handily use the information inserted in context data streams to implement the desired behavioral variations. The programmer has only to care about the definition of layers, with statements, partial methods, etc., for the entities whose normal behaviors must be dynamically adapted.

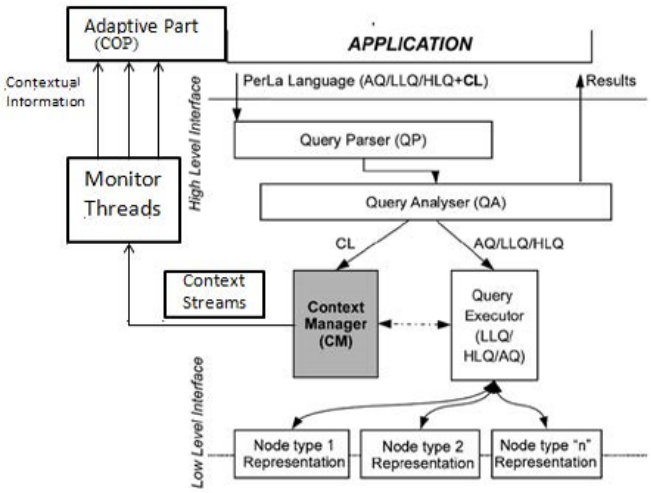


Figure 13: The system architecture

Fig. 13 shows the resulting architecture.

This solution can be adopted in large and complex applications, where a considerable number of modules, belonging to different parts, need adaptations; however, it does not provide a real integration between the two treated actors. In fact, it is necessary to design dedicated components in order to to provide the proper contextual data provided by PerLa to COP which, in turn, performs the behavioral variations for the application. These components form a kind of intermediate layer

```
1
2  CREATE CONTEXT SmokeMonitoring
3  ACTIVE IF Location = 'office' AND Smoke = 'persistent'
4  REFRESH EVERY 1h
5
6  ON ENABLE (SmokeMonitoring) :
7  SELECT smoke
8  SAMPLING EVERY 10 m
9  EXECUTE IF EXISTS (smoke)
10 LOAD COP CONTEXT SmokeRisk
11 SET PARAMETER speed = 0
12
13 ON DISABLE (SmokeMonitoring) :
14 DROP SmokeMonitoring
15 DROP COP CONTEXT SmokeRisk
```

Listing 14: The integrated COP-PerLa approach

that separates the information source and the adaptive part of the application.

In Listing 14, the new clauses *LOAD COP CONTEXT/DROP COP CONTEXT* have been introduced, in order to tell the system to activate/deactivate the related COP context (in this example, the COP code refers to Listing 13). The two clauses allow PerLa to execute external code (e.g., Java code) implemented following the COP paradigm, incrementing the whole PerLa CL and PerLa QL expressive power, introducing in them the possibility to perform operations that a SQL-like only approach cannot execute (and often also cannot express)

The activated COP context is then responsible of the relevant layer activation (e.g. *SmokeLayer*) and of managing all the related threads and procedures, while PerLa only manages the COP context activation/deactivation and handles the switch among different contexts. Notice that the COP context activation is not the only action performed by the system, since it firstly sends the air conditioner the "stop" (setting its speed to "0") command using the default PerLa *SET PARAMETER* clause.

References

Cristiana Bolchini, Carlo Curino, Giorgio Orsi, Elisa Quintarelli, Rosalba Rossato, Fabio A. Schreiber, and Letizia Tanca. And what can context do for data? *Commun. ACM*, 52(11):136–140, 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1592761.1592793.

Emanuele Panigati, Angelo Rauseo, Fabio A. Schreiber, and Letizia Tanca. Aspects of pervasive information management: An account of the green move system. In *IEEE 15th International Conference on Computational Science and Engineering, CSE 2012, Paphos, Cyprus, December 5-7, 2012* Panigati et al. [2012], pages 648–655. doi: 10.1109/ICCSE.2012.93. URL `http://doi.ieeecomputersociety.org/10.1109/ICCSE.2012.93`.

Fabio A. Schreiber, Letizia Tanca, Romolo Camplani, and Diego Viganò. Managing and using context information within the perla language. In *Sistemi Evoluti per Basi di Dati - SEBD 2011, Proceedings of the Nineteenth Italian Symposium on Advanced Database Systems, Maratea, Italy, June 26-29, 2011* Schreiber et al. [2011], pages 111–118.

Fabio A. Schreiber, Romolo Camplani, Marco Fortunato, Marco Marelli, and Guido Rota. Perla: A language and middleware architecture for data management and integration in pervasive information systems. *IEEE Trans. Software Eng.*, 38(2):478–496, 2012a.

Fabio A. Schreiber, Letizia Tanca, Romolo Camplani, and Diego Viganó. Pushing context-awareness down to the core: more flexibility for the PerLa language. In *Electronic Proc. 6th PersDB*, pages 1–6, 2012b.