POLITECNICO DI MILANO
Facoltà di Ingegneria
Scuola di Ingegneria dell'Informazione

Design and development of an asynchronous data access middleware for Pervasive Networks: the case of PerLa

Relatore: Ch.mo Prof. Ing. Fabio Alberto Schreiber
Correlatore: Ing. Emanuele Panigati

Tesi di laurea di:
GUIDO ROTA
Matr. 819938

Anno Accademico 2013 - 2014

Sommario

La presente tesi descrive le fasi di progettazione e sviluppo di un middleware per la gestione dei dati generati da Sistemi Pervasivi. L'obiettivo principale di questo software consiste nel fornire un'interfaccia di alto livello finalizzata all'estrazione di informazioni da un Sistema Pervasivo, vale a dire una rete eterogenea composta da dispositivi di misura ed attuazione di varia natura. L'architettura proposta si basa su un paradigma di programmazione ad eventi che permette una gestione asincrona dei flussi dati generati dalla rete di sensori controllata.

Abstract

This thesis describes the design and development of a data management middleware for Pervasive Systems. The main goal of this software consists in the definition of a high-level abstraction layer that can be used to collect information from a Pervasive System, i.e., a heterogeneous network composed of sensing and actuating devices with different characteristics. The proposed architecture is based upon an event-driven programming paradigm, which enables an asynchronous management of the data streams generated by the underlying sensing network.

Contents

1 Introduction **1**

1.1 Wireless Sensor Networks and beyond 2

1.2 Data management in Pervasive Systems 3

2 The PerLa System **6**

2.1 A brief history of PerLa 6

2.2 The Classic Middleware Architecture 8

2.2.1 Main goals and operating principles 8

2.2.2 The Functionality Proxy Component 9

2.2.3 Plug & Play device addition 12

2.3 The PerLa Query Language 13

2.3.1 The Data Management section 14

2.3.2 Sampling section 14

2.3.3 Conditional Execution section 15

2.3.4 Termination Condition section 15

2.3.5 Query examples 16

2.4 Context management 17

3 The New PerLa Middleware **18**

3.1 Design goals 18

3.2 Overview of the New Middleware Architecture 19

3.2.1 The New FPC 20

3.2.1.1 Channel 21

3.2.1.2 Mapper . . . 22
3.2.1.3 Scripts and Operations . . . 22
3.3 FPCFactory . . . 24
3.4 Asynchronous interaction paradigm . . . 27

4 In-depth component description 31
4.1 Communicating with Channels . . . 31
4.1.1 Instantiating new Channels . . . 33
4.1.2 IORequest management . . . 36
4.1.3 Handling asynchronous I/O operations . . . 40
4.2 Handling data . . . 43
4.2.1 The Message interface . . . 43
4.2.2 Working with Messages: the Mapper interface . . . 46
4.2.3 Creating Mappers and defining Message structures . . . 49
4.2.4 Managing multiple message types . . . 51
4.3 Data management: Scripts . . . 53
4.3.1 Anatomy of a PerLa Script . . . 56
4.3.1.1 `var` instruction . . . 58
4.3.1.2 `set` instruction . . . 58
4.3.1.3 `append` instruction . . . 59
4.3.1.4 `submit` instruction . . . 60
4.3.1.5 `stop` instruction . . . 61
4.3.1.6 `error` instruction . . . 62
4.3.1.7 `put` instruction . . . 63
4.3.1.8 `emit` instruction . . . 64
4.3.1.9 `if` control structure . . . 65
4.3.1.10 `foreach` control structure . . . 66
4.3.2 Script Engine architecture and execution model . . . 67
4.4 Putting it all together: the FPC . . . 70
4.4.1 Accessing device features . . . 71
4.4.1.1 Get Operation . . . 71
4.4.1.2 Set Operation . . . 72

4.4.1.3 Periodic Operation 72
4.4.1.4 Async Operation 74
4.4.2 The FPC interface 75
4.4.3 FPC Factory 78
4.4.4 Registry 81

5 Conclusions 83
5.1 Future work 85
5.1.1 Implementation of new plugins 85
5.1.2 Alternative Device Descriptor forms 86
5.1.3 Distributed PerLa 87

Bibliography 88

Appendix A Complete XML Device Descriptor examples 91
A.1 Example 1 91
A.2 Example 2 94
A.3 Example 3 97

Chapter 1

Introduction

Computing devices permeate every aspect of our life. PCs, tablets, smartphones, identification badges, credit cards, smart-watches, wearable gadgets, traffic cameras, digital fitness bands, and personal medical devices like pacemakers or insulin injectors are only a few of the tools that we use every day, more or less consciously, to produce and consume information.

As the number of devices and services that surround ourselves increases, so does the level of mutual cooperation that we expect from them. We know that our smartphone will automatically show us the weather for our current location, sometimes even for our hometown when we are away (How does it know where I live? Did I ever tell it?). Navigation apps guide us through different itineraries at different times of the day, depending on current and expected traffic conditions. Outbreaks of the most common viral diseases can be traced and monitored by analysing what people are searching on the web and correlating that data with other sources like hospital records.

We have grown so accustomed to the tight level of integration between different services and information sources, that behaviours similar to those presented above are nowadays expected. User requirements have heightened, and products can fail to gain traction if they don't exploit the data that is available around us in

new and innovative ways; different computing devices and services must discover themselves and make mutual use of the information that they produce or consume, meshing together in what is called a Pervasive System.

Firstly envisioned by Mark Weiser [17], Pervasive Systems are connected networks of independent and heterogeneous devices, whose ultimate goal is to assist people in a way that is effectively invisible to the final user. They are the result of a post-PC era, where scores of computing gadgets are disseminated in our surroundings, enhancing our capabilities to sense the world and providing ubiquitous access to information.

From a software and hardware perspective, a Pervasive Systems is a rich and varied environment, hosting myriads of different network protocols, data formats, and incompatible CPU architectures. Tapping the ever increasing stream of data produced in such a heterogeneous context, and using it to build advanced and connected products, can easily become a daunting task. Since Weiser's seminal paper, several endeavours have explored different techniques for simplifying the task of building and designing Pervasive Systems, many of which stemmed from the research on Wireless Sensor Networks (WSN).

1.1 Wireless Sensor Networks and beyond

As suggested by the name, Wireless Sensor Networks (WSNs) are networks of wirelessly connected devices called nodes or motes, that are able to measure or detect physical properties from their surrounding environment. Although the original acronym only references the sensory features of such systems, current usage of the term WSN is commonly extended to include devices with actuation capabilities.

Wireless Sensor Networks provide a low-cost and effective solution for monitoring physical phenomena: several inexpensive nodes may be scattered around the area of observation, without requiring an explicit configuration or a wired communication infrastructure. Data is autonomously routed from the point of origin to the

interested consumers, where it is usually aggregated, analysed and presented to the user. Flexibility and ease of deployment make WSNs the ideal candidate for a plethora of applications, covering home automation, theft prevention systems, healthcare, control of environmental hazards and monitoring of production lines.

The rapid increase in popularity of WSNs, coupled with the intrinsic difficulties of working in a variegated software and hardware environment, spurred the development of a wide variety of frameworks, middlewares, and ad-hoc programming languages designed for providing an easy way of access to the functionalities offered by networks of sensing and actuating devices and Pervasive Systems alike.

This thesis describes the design and implementation of one of these software systems, an asynchronous data access middleware for Pervasive Networks dubbed *New PerLa Middleware*. Its history, as will be shown in chapter 2, is strongly intertwined with the PerLa Query Language [15], a declarative SQL-like language for collecting data from WSNs and other distributed sensing networks. However, before delving into the substance of this middleware, the next section will provide a short survey on the different approaches proposed in literature for the management of Pervasive Systems.

1.2 Data management in Pervasive Systems

Early experiences in managing sensing networks were based on ad-hoc systems that provided bespoken solutions to specific applications. These approaches were usually built using proprietary hardware and highly customized software architectures, whose design was ultimately concerned with the implementation of a limited and well-defined series of task-oriented requirements. As shown in [9] [14] [18] [10] these highly personalized systems are poorly reusable, since their architecture usually fails to provide a clear separation of concerns between the pure data access mechanisms employed to control the underlying sensing network, and the particular behaviour required by the specific application. It soon became clear that this strategy could not become a viable model for a simple and effective development

of pervasive applications, as all the effort and expertise put in the implementation of such systems could not be efficiently exploited and shared in new projects.

TinyDB [13] is one of the first endeavours that tried to provide a generic abstraction suitable for the development of sensing applications based on Pervasive Systems. A WSN managed by TinyDB is in fact presented to the final users as a vast streaming database controlled through SQL-like queries, which are appropriately interpreted and distributed to the single nodes of a pervasive system in order to specify a globally coordinated behaviour. Despite the general high-level abstraction employed by this system, the execution of TinyDB queries requires all nodes in the sensing network to be running the TinyOS [11] embedded operating system, therefore limiting the variety of devices which can be managed using this approach.

Similarly to TinyDB, DSN [4] aims at providing a database-like abstraction of an entire Pervasive System, which can be queried and controlled using a Datalog dialect dubbed Snlog. A comparable "WSN as a database" abstraction, as will be shown in chapter 2, is a characteristic feature of the PerLa System as well.

Other projects tried to approach the problem of handling a Pervasive System by shifting the focus on the management of highly heterogeneous networks of sensing devices. SWORD [8] tries to achieve this goal by providing a central infrastructure that can be used to monitor events and signals collected from a distributed mesh of nodes. Unfortunately, its XML-over-HTTP messaging system considerably reduces the variety of devices that can be integrated. GSN [2], on the other hand, is a Java Middleware based on the concept of *Virtual Sensor*, a high-level software component aimed at providing a common interface that can be used to collect information generated by a single device of a sensing network. Virtual Sensors are created by means of a declarative XML descriptor, and can be interfaced with the remote endpoints through a specific device driver dubbed *wrapper*. Unfortunately GSN does not provide any mechanism for automating the deployment of new Virtual Sensors.

The TinyREST [12] project attempts to provide a RESTful [6] data access interface

to each node available in a Pervasive System. This goal is achieved through a server infrastructure that performs a mapping between HTTP methods and various Input/Output operations performed by the devices of a sensing network. As a result, an HTTP GET request can be used to sample a physical phenomena, whereas a PUT operation allows users to command actuators or set variables on remote devices. Its reliance on the TinyOS makes it vulnerable to the same critiques made to TinyDB.

Contiki [5] is a lightweight and portable operating system specifically designed for memory-limited devices. Differently from TinyOS, Contiki tries to foster the same programming model available to desktop machines. Its kernel, whose footprint may vary between 10 and 30 KB of RAM, provides in fact a fully functional IPv4 and IPv6 networking stack, a multi-thread concurrency mechanism based on protothreads and an over-the-air programming mechanism. Differently from the other projects discussed in this section, Contiki does not provide any high-level abstraction for accessing the information produced by a Pervasive System.

Chapter 2

The PerLa System

2.1 A brief history of PerLa

PerLa is a software infrastructure for data management and integration in Pervasive Information Systems. Its development began in 2005 at Politecnico di Milano, with a thesis by Marco Marelli and Marco Fortunato entitled "*A Declarative Language for Pervasive Systems*" [7]. With this document the two authors laid the foundations for a completely declarative, SQL-like language that could be used to gather information from Pervasive Systems and Wireless Sensing Networks alike. Though their work primarily focussed on defining the syntax and semantics of the PerLa language, Marelli and Fortunato went on to propose a reference software architecture that could support the execution of PerLa data collection queries.

In their first design they envisioned the possibility of creating a *Device Access Layer*, whose goal was to conceal all the idiosyncratic features of a Pervasive System, and provide a homogeneous data access interface that could be used as a thought device to support the first development stages of PerLa. The principal element of this initial software architecture was the *Logical Object*, a virtualization module that provides a uniform API for accessing the functionalities of a single device in the sensing network; this germinal design evolved during the course of

the following years into what would later be known as *PerLa Middleware*.

The PerLa Middleware [15] was primarily concerned with providing an actual implementation to the Logical Object abstraction, a goal that was achieved with the creation of the Functionality Proxy Component (`FPC`). The `FPC` is a Java entity that reifies all concepts embodied by the Logical Object, with particular emphasis on the ability to abstract the peculiar features of a single sensing device through a common and uniform programming interface. The PerLa Middleware, however, was more than a simple implementation of the Logical Object, as it provided a *Plug & Play* system for the autonomous creation of `FPC` objects, as well as an initial release of the PerLa Query Executor component.

The development of the PerLa Middleware has been a collaborative effort that involved multiple students and several years of work. Therefore, the product that ensued is the sum of all contributions made by different people, that, at one time or another, put their minds and hearts at the design and implementation of the system. While such development approach allowed PerLa to thrive and mature rapidly, since many intellects had the possibility to contribute with their innovative ideas, it also meant that the growth of the system has been inconstant and, at times, confusing. It is under these premises that, in early 2014, the Middleware underwent a complete redesign, whose primary intentions were to consolidate the main programmatic API (Application Programming Interface), improve performances, and further advance some of the defining characteristics of its architecture. The document you are reading is an account of this recent endeavor, and contains a description of the past, present and foreseeable future of the PerLa System.

The remaining sections of this chapter provide an overview of PerLa prior to the current redesign, starting with the previous Middleware architecture — also known as Classic PerLa Middleware — and ending with a short digression towards Context management in Pervasive Systems. This same chapter will describe the main features of the PerLa Query Language, which as of today still is the interface of choice for using PerLa. Chapters 3 and 4 contain an in-depth description of

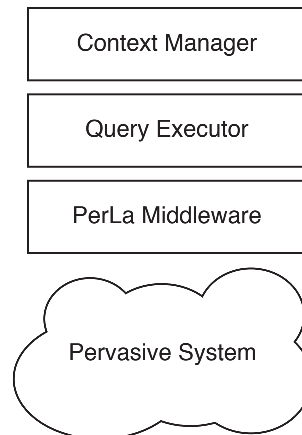

Figure 2.1: A macro component view of the PerLa System.

the New Middleware architecture, which should be of concern to any programmer interested in developing PerLa Plugins or connecting new types of sensing devices. Finally, chapter 5 wraps up the work illustrated in this thesis, and provides an outlook on the prospects for PerLa's future.

2.2 The Classic Middleware Architecture

2.2.1 Main goals and operating principles

The PerLa Middleware is a complex software based on the Java platform. Its development focussed primarily on the design and implementation of the following features:

- data-centric view of Pervasive Systems;
- homogeneous high level interface to heterogeneous devices;
- support for highly dynamic networks (e.g., wireless sensor networks);
- minimal coding effort for new device addition.

The remaining of this chapter will show how the implementation of the `FPC` component and the development of a Plug & Play device addition mechanism contributed

to achieving the aforementioned goals. The reader is strongly encouraged not to skip these following sections, as most of the concepts hereby introduced will be subject of further discussion in chapter 3, where they are going to be matched alongside their novel counterparts.

2.2.2 The Functionality Proxy Component

As previously stated, the principal element of the PerLa Middleware, both in its classic and redesigned incarnations, is the `FPC`. Its primary function consists in providing a consistent and homogeneous interface that can be used by high-level components of the PerLa System to access all functionalities of a Pervasive System, without requiring any knowledge of the underlying hardware layer.

All data elements accessible through an `FPC` are abstracted using the concept of *Device Attribute*, namely a single piece of information produced or consumed by a node in the Pervasive System. By design, device attributes are not tied to a particular technology, and can therefore be used to represent any primitive value, regardless of the method employed for its generation (physical sampling, read from main memory, etc.). Device Attributes are one of the defining aspects of the PerLa Middleware, and greatly contribute to the creation of a data-centric abstraction that can be used to access all features of a sensing network; any operation performed through the `FPC` interface is in fact specified in terms of reading or writing a specific set of attributes.

In the Classic PerLa Middleware, the physical connection between an `FPC` and its remote device is entirely managed by the `Channel Manager` (see figure 2.2). This singleton component is responsible for the creation of `Virtual Channels`, abstract network interfaces that conceal the specific technologies required to hold a communication link between two endpoints of a Pervasive System. By means of the `Channel Manager`, `FPC`s and physical devices can exchange information transparently, mutually ignoring all the architectural differences that may exist between them. The achievement of a truly universal communication system, however, is

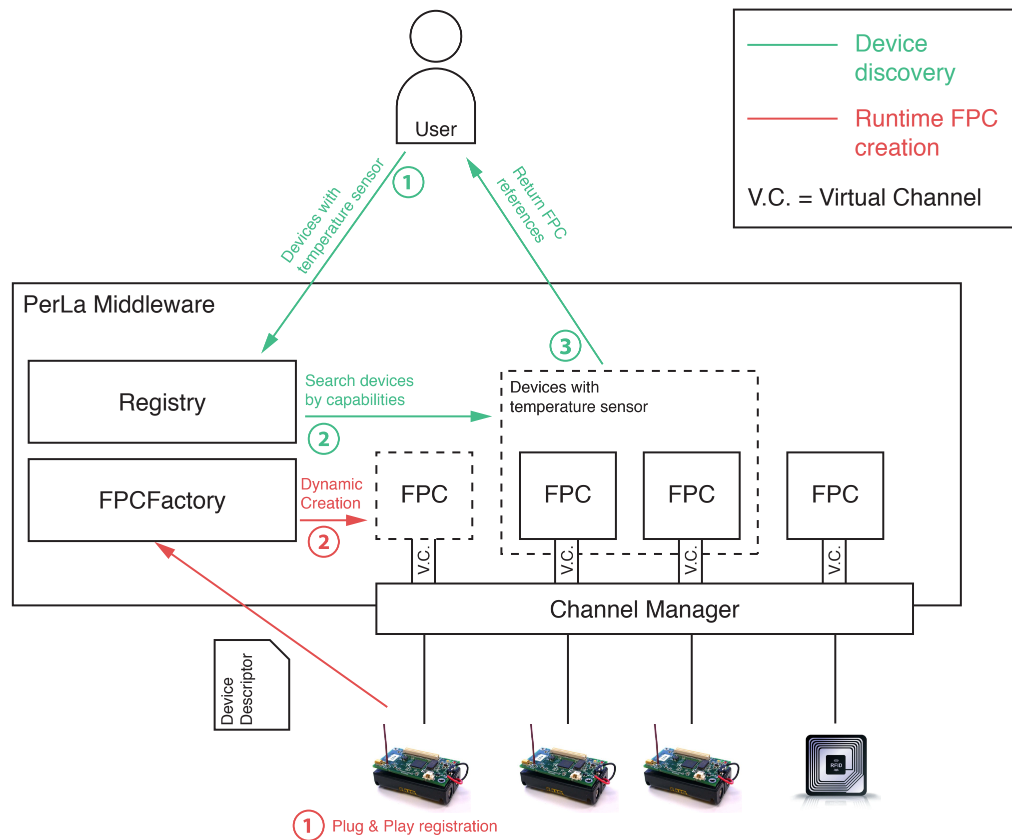

Figure 2.2: The Classic PerLa Middleware architecture

severely limited by the capabilities of the `Channel Manager` itself, as the choice of protocols available in the Classic Middleware is strictly restricted to those that the original PerLa Developers hard-coded into this component.

As shown in figure 2.3, the internal structure of the Classic `FPC` is mainly composed of a `Marshaller`, an `Unmarshaller`, and a selection of Java objects representing the messages which can be exchanged with the remote endpoint. This early design is enough to implements a simplified data collection mechanism, based on the assumption that every device attribute can be mapped to exactly one message field. Under the Classic Middleware, a typical data collection request for a well-defined set of attributes would then be executed as follows:

1. Starting from a user's request, and leveraging the one-to-one relationship be-

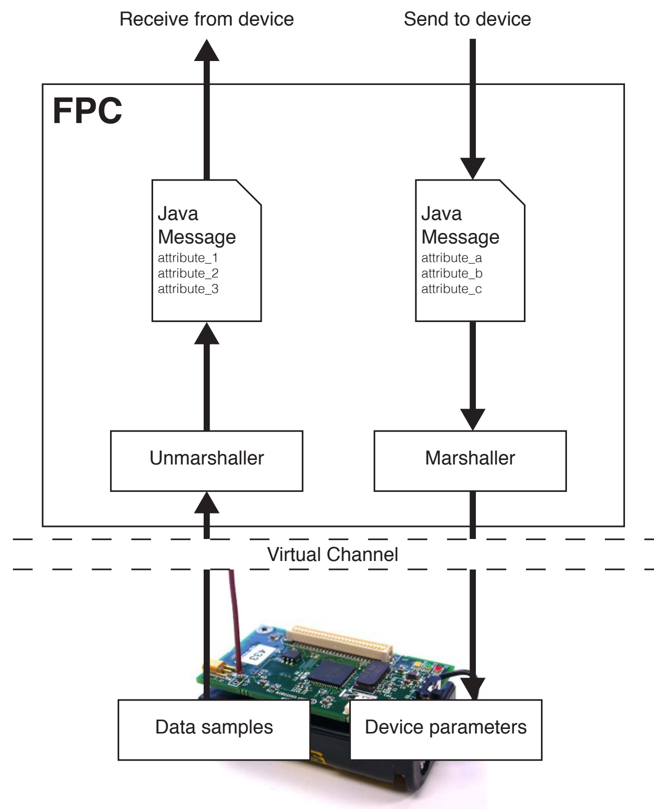

Figure 2.3: The Classic FPC design

tween attributes and message fields, the `FPC` compiles a list of data structures which are to be collected from the remote device;

2. The remote device starts streaming data to the `FPC`. This information is unmarshalled into a high-level Java message, according to the directives specified inside the Device Descriptor (see section 2.2.3);
3. All message fields containing relevant information for the user are read in order to produce an output record.

A similar process, making use of the `Marshaller` component instead of the `Unmarshaller`, and with a reversed data flow, allows the `FPC` component to send configuration parameters towards its controlled device.

2.2.3 Plug & Play device addition

Sensing nodes can be added to a running instance of the PerLa Middleware through a *Plug & Play* device registration mechanism that allows the discovery and configuration of new devices without the need for direct user interventions. This operation is performed at runtime by the `FPCFactory`, a software component tasked with creating new `FPC` objects from a blueprint document called *Device Descriptor*. PerLa-enabled devices are programmed to send their XML descriptor at startup; this allows them to be readily available for use, as their controlling `FPC` object is dynamically instantiated by the `FPCFactory` upon reception of the Device Descriptor. Figure 2.2 contains a graphic portrayal of this entire process.

Figure 2.4: Physical to logical attribute mapping in the Classic PerLa Middleware. The annotations added to the Java Message allow the `Marshaller` and `Unmarshaller` components to encode and decode the information exchanged with the remote device.

The information required to create all internal components of an `FPC`, along with the corresponding attribute-message mappings and `Marshaller`/`Unmarshaller` directives, are extracted from the Device Descriptor. Figure 2.4 contains a typical example of such file, which highlights the interdependence that exists between C-Struct declarations used in the sensing device, corresponding Device Descriptors, and the resulting Java classes created by the `FPCFactory`.

All `FPC` objects created by the `FPCFactory` are catalogued and inventoried in the

`Registry`, a simplified main-memory database that contains a complete directory of all devices connected to the PerLa Middleware. By means of the `Registry`, users can sift through all nodes of the network, and select those that best suit their current computational needs. As will be shown in the next section, this component is fundamental for the implementation of the `EXECUTE IF` query clause.

2.3 The PerLa Query Language

The PerLa Query Language is a declarative, SQL-like language for interacting with Pervasive Systems. A sensing network managed by PerLa is abstracted as a large table in a streaming database, whose columns correspond to specific `Attributes` that can be retrieved from the devices connected to the Middleware. This generalization allows final users to glean information out of a Pervasive System without dealing with all the complications that stem from managing the quirks of every single sensing node, as the intricate mesh of available data sources is completely hidden by the database abstraction.

The PerLa Query Language is designed to be simple and easy to use. Its core syntax is compact and reminiscent of other well-known database-oriented languages as SQL, and provides a uniform interrogation mechanism that enables the collection of data elements regardless of their origin; information may be sampled from a physical phenomena, read from the memory of an endpoint device or extracted from a web service: whatever the source, the PerLa query pattern is always the same.

The reminder of this section will provide an overview of the main syntactic and semantic features of the PerLa Query Language, along with two examples excerpted from real-world use cases.

2.3.1 The Data Management section

Introduced by the `SELECT` clause, this portion of the PerLa Query Language should immediately result familiar to every developer acquainted with classic SQL. This clause achieves two purposes: first, it defines which data elements (specifically, which data `Attributes`) are to be collected from the Pervasive System; second, it indicates the operations and computations that must be performed on the information being extracted.

The need to manage a theoretically infinite stream of data elements coming from the sensing network required the development of a custom syntax for aggregate operations. Differently from standard SQL aggregates, which always operate on a finite set of elements, PerLa aggregates must cope with an ever-flowing stream of records, and thus require users to specify the scope of their intended computations. This is achieved through a duration expression, a mandatory parameter that complements the aggregation clause by limiting the number of records to be processed to a limited amount. Duration expressions can be specified using two different methods: a time-based syntax, that allows users to define the aggregation scope in terms of time windows (`SELECT AVG(TEMP, 10 SECONDS)`), and a record-based syntax, that clearly indicates the number of records to be used for the computation (`SELECT AVG(TEMP, 30 SAMPLES)`).

2.3.2 Sampling section

The Sampling section can be used to specify how and when the data `Attribute`s requested with the `SELECT` statement are to be extracted from the network nodes. There are two different operating modes, both introduced by the `SAMPLING` clause. *Time-based* sampling is employed to collect data at periodic intervals; in this mode, the sampling frequency is specified by means of an `IF-EVERY` syntax, which enables users to specify different expression-guarded sampling periods (see listing 2.1 for an applicative example). On the other hand, the *Event-based* sampling mode allows the acquisition of a data sample each time a specific event is fired.

```
SAMPLING
    IF temperature < 50 EVERY 10 MINUTES
    ELSE IF TEMPERATURE >= 50 EVERY 1 MINUTES
```

Listing 2.1: An example of time-based sampling, which shows how the sampling frequency can be increased as the monitored phenomenon evolves.

2.3.3 Conditional Execution section

Introduced by the `EXECUTE IF` clause, this query section contains a boolean expression that every sensing device must satisfy in order to be considered as a candidate data source, and it's often employed when the user requires its query to be executed on nodes with well-defined capabilities. This section is optional, and its omission implies that the PerLa query must be executed on every device of the sensing network. An `EXECUTE IF` statement can be optionally complemented by a `REFRESH` clause, which specifies how often the execution condition is re-evaluated to update the list of nodes involved in the evaluation of a query.

2.3.4 Termination Condition section

The Termination Condition is an optional clause that can be used to stop a query after a specific of amount of time (`TERMINATE AFTER 1 DAY`) or number of executions (`TERMINATE AFTER 10 SELECTIONS`). This behaviour is useful when perform a one-shot query, or when the monitoring period is known a priori.

2.3.5 Query examples

The following query initiates a temperature sampling operation on all temperature sensors located in room number three. New data readings are collected by the minute, as specified in the SAMPLING clause; however, new output records are created every 5 minutes, as indicated in the EVERY statement that guards the entire data management section. Thanks to the MAX aggregation expression, each record produced by this query contains the maximum temperature value collected in the previous 10 minutes of sampling.

```
1 CREATE OUTPUT STREAM Table (Temperature FLOAT) AS:
2 EVERY 5 MINUTES
3 SELECT MAX(temp, 10 MINUTES)
4 SAMPLING
5   EVERY 1 MINUTES
6 EXECUTE IF EXISTS(temp) AND EXISTS(room) AND room = 3
```

This second example illustrates how the PerLa Language can be used to collect information in response to an event. As can be seen from the TERMINATE AFTER clause, this one-shot query terminates as soon as the first record is produced. Its single output record contains the number of times the RFID with identifier 0xDF445A was scanned in the last 10 minutes.

```
1 CREATE OUTPUT STREAM Table (rfid STRING, counter INTEGER) AS:
2 EVERY 10 MINUTES
3 SELECT lastReaderId, COUNT(*, 10 MINUTES)
4 SAMPLING
5   ON EVENT lastReaderChanged
6 EXECUTE IF ID=[0xDF445A]
7 TERMINATE AFTER 1 SELECTIONS
```

2.4 Context management

The Context Management layer [16] 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, [3]), 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.

Chapter 3

The New PerLa Middleware

3.1 Design goals

Redesign of the PerLa Middleware began in early 2014 with the intention to consolidate the existing software infrastructure and refine its main features. This process set in motion a critical analysis of the existing architecture, which led to a complete rethinking of many of the Middleware's distinctive components and processes, like the `FPC` and its Plug & Play device addition mechanism. As will become clear, not all components required the same amount of changes; some varied much, other less, while others practically remained unchanged (for example, the `Registry`).

The main driving force that pushed this re-engineering endeavour was the need to strengthen the major Middleware APIs, and elegantly integrate all isolated contributions made by past PerLa developers into a single, coherent product. Furthermore, this redesign proved to be an excellent opportunity for expanding the capabilities of the PerLa Middleware itself, and to better prepare it to cope with a constantly evolving Pervasive Environment where web-services and RESTful data sources are ever more present. This last requirement, above everything else, has been crucial to the development of the final Middleware architecture described in

this thesis, as it enforced a precarious balance between two seemingly incompatible necessities, namely the need to interface with low-level, resource-limited sensing nodes and with high-power web services alike.

In summary, the goals pursued during the re-engineering of the PerLa middleware amounted to:

- Consolidate all programming interfaces exposed to the final users, with a special regard to the `FPC` and the Device Descriptor;
- Integrate and coalesce the internal programming interfaces employed to interconnect the various Middleware modules using an asynchronous, event-driven paradigm;
- Improve the expandability of the system by introducing a modular design composed of pluggable and independent software units;
- Identify and solve potential bottlenecks that could prevent the Middleware from scaling gracefully under load.

The remainder of this section presents an overview of the New PerLa architecture; its intended objective is to convey the major differences that exist between the two Middleware designs, while explaining how they contribute to achieving the aforementioned goals. An in-dept description of all modules described in the following is available in chapter 4.

3.2 Overview of the New Middleware Architecture

As illustrated in the previous chapter, the PerLa Middleware is responsible for managing the lifecycle of all devices connected to the PerLa System, 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. The most prominent trait of the `FPC` is its interface, an API that allows PerLa users to interact

with the sensing network through a compact set of hardware-agnostic communication primitives reminiscent of classic Java getter and setter methods. Use of this interface neither requires knowledge of the sensing network, nor of the device that will ultimately perform the requested operation.

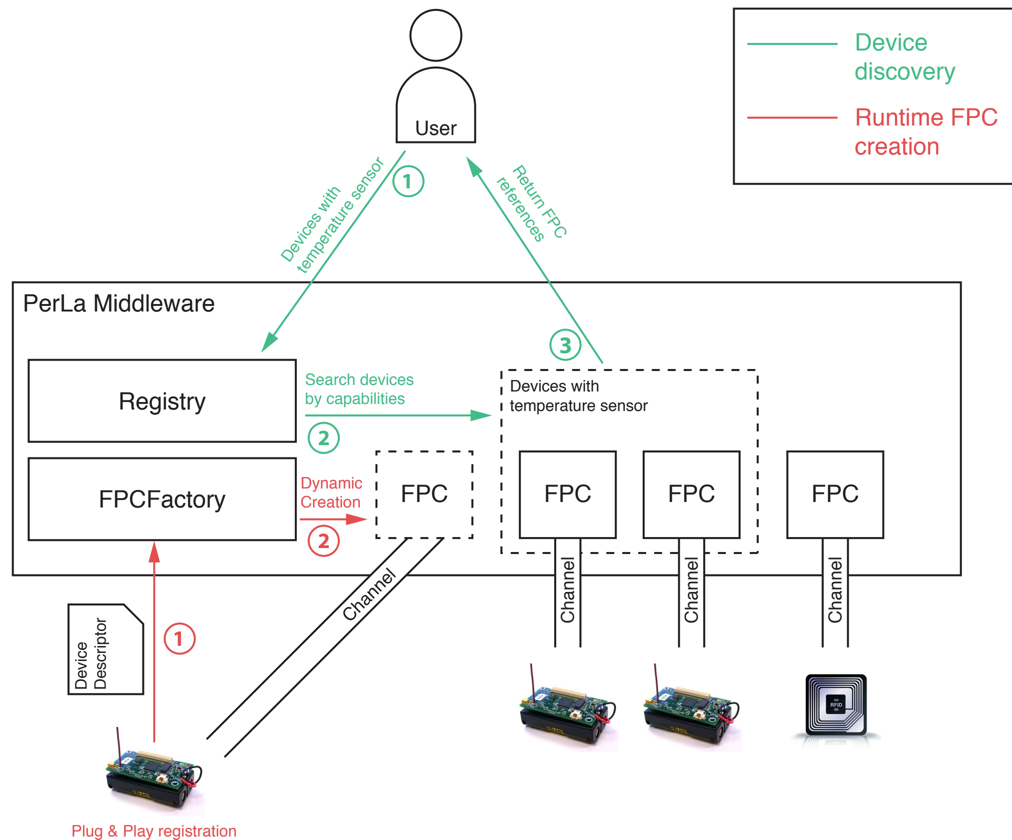

Figure 3.1: The New PerLa Middleware architecture

3.2.1 The New FPC

Differently from the Classic PerLa Middleware, the New `FPC` is formed from the composition of various independent software units, each of which is responsible for the management of a single aspect of the interaction with the remote device (see figure 3.2). This new modular design was chosen to further promote reusability

and foster future expandability through composition of independent objects. The remainder of this section contains a summary of all modules that compose the new FPC architecture.

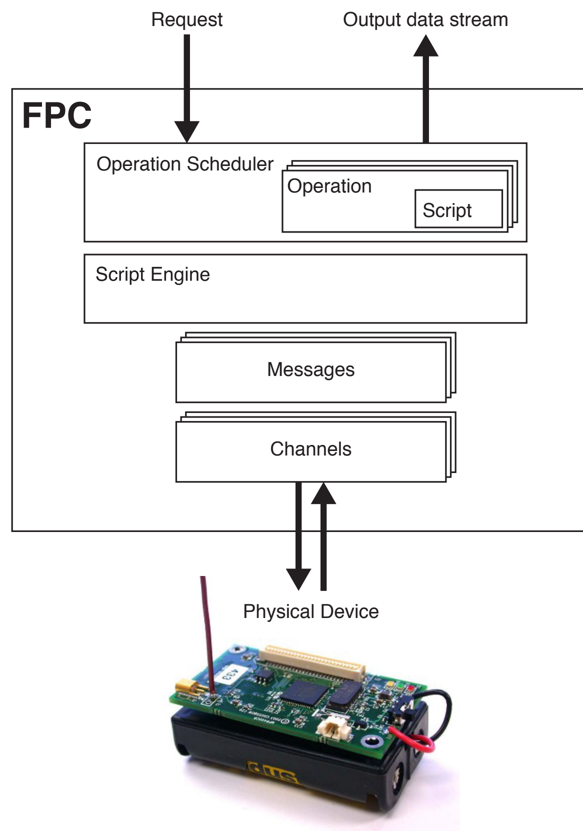

Figure 3.2: Internal structure of the New FPC.

3.2.1.1 Channel

Channels are software components capable of performing I/O operations. They are commonly employed to manage the communication between PerLa and the devices of a Pervasive System, and can be thought as a complete substitute of the Channel Manager/Virtual Channel component pair. Unlike their counterpart in the Classic architecture, Channels are settled inside the boundary of an FPC. This change of location has two important consequences: first of all, it allows each

`FPC` to make use of multiple data transmission technologies for the exchange of information with the controlled endpoint; second, it enhances the modularity of the entire PerLa Middleware, as new communication systems and protocols can be introduced without modifying any existing `Channel` implementation.

Every `Channel` is bundled with a collection of `IORequest` objects, which are employed to initiate specific I/O tasks on the sensing nodes. For example, the `HTTPChannel` — a `Channel` implementation of the HTTP protocol — can be used with four different `IORequest`s, one for each of the principal HTTP methods (GET, POST, PUT and DELETE).

3.2.1.2 Mapper

A software module for marshalling and unmarshalling data. `Mapper`s allow the `FPC` to interpret byte streams received from a communication `Channel`, and to serialize high-level data structures prior to transmission. They perform as a more flexible alternative to the fixed `Marshaller-Unmarshaller` components found in the Classic `FPC` implementation.

It is important to note that every `Mapper` is a discrete component responsible for managing a specific `Message` format. Therefore, similarly to what already seen for the `Channel` object, a single `FPC` may employ different `Mapper` objects tasked with managing a well-defined set of data structures exchanged with the remote device. This feature is in stark contrast with the Classic Middleware design, which could only handle a single data format per device.

3.2.1.3 Scripts and Operations

`Scripts` and `Operations` are new additions to the PerLa Middleware. Their primary duty consists in binding high-level data requests to native processing tasks performed on the remote device.

`Scripts` are interpreted programs written using the *PerLa scripting language*, a

simple imperative programming language designed for interacting with sensing nodes and for mapping low-level data structures to high-level `Attributes` generated by an `FPC`. `Scrips` are executed by a singleton component denominated `Script Engine`.

`Operations`, on the other hand, are data management entities that incorporate one or more `Scripts` dedicated to a specific purpose. The New Middleware architecture distinguishes between four `Operation` types, corresponding to different data collection or transmission strategies:

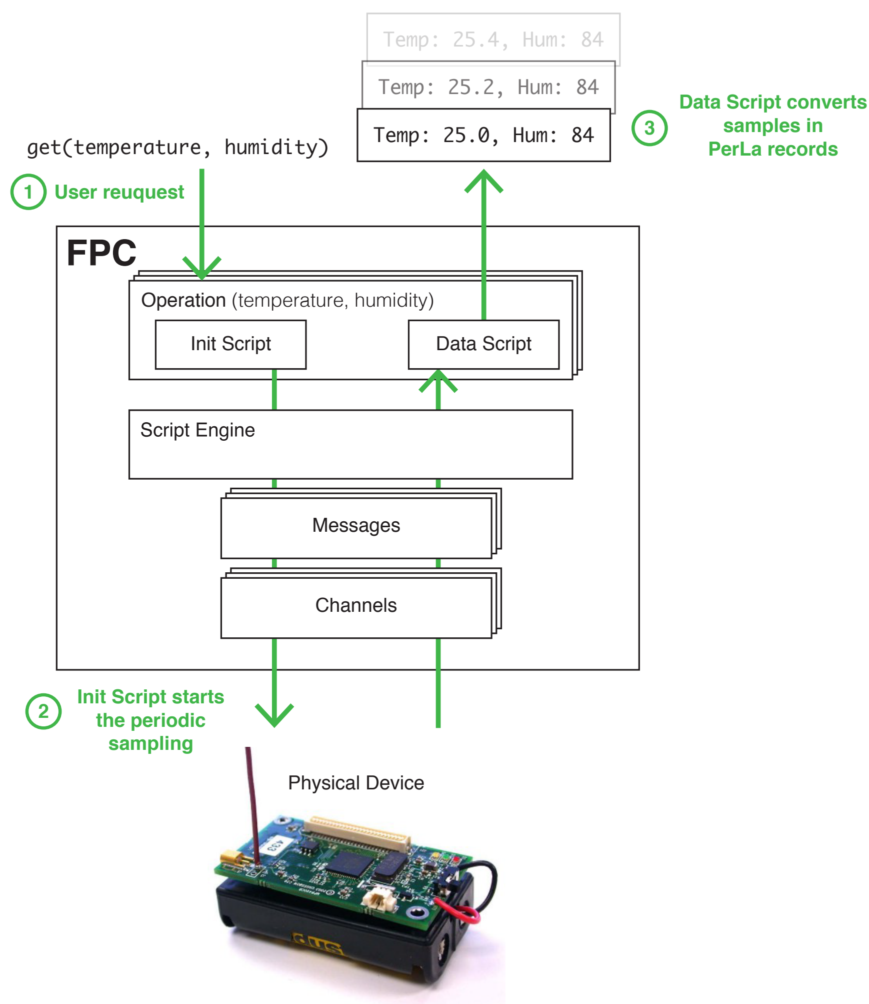

Figure 3.3: The execution of a user request in the New `FPC`

- `Get Operation:` retrieves a single sample from the remote device;
- `Set Operation:` transmits data to the remote device;
- `Periodic Operation:` manages the periodic collection of data samples generated by an endpoint of the Pervasive Network;
- `Async Operation:` handles the reception of asynchronous events emitted by the controlled node.

It is important to note that every `Operation` is associated with the set of Device `Attribute`s that can extract or set through it (see 3.3). Therefore, under the New Middleware architecture, a typical data collection request is executed as follows:

1. The `FPC` selects an `Operation` that can be used to adequately retrieve the `Attribute`s requested by the user;
2. The `Operation`'s initialization `Scripts`, if any, are executed in order to configure the remote device and commence the collection of data;
3. Every data sample received from the `FPC` is processed by a `Script`, as per `Operation` instructions, which is tasked with unmarshalling the information sent by the endpoint and create a new output record.

3.3 FPCFactory

The new `FPCFactory` is a modular software entity composed of multiple factory components dedicated to the creation of a specific `FPC` module. Its design is a significant departure from the original monolithic factory structure, and allows final users to easily expand the base capabilities of the PerLa Middleware without modifying its original source code. This modular architecture, formally called `FPCFactory` Plugin System, is a direct implementation of the *Open/Close* principle: the `FPCFactory` is open for extension, as its functionalities can be expanded by adding new plugins, but closed for modification, since the addition of a new sub-factory modules does not require any change to the Middleware's source code.

At the current state of implementation, the PerLa Middleware supports three dif-

ferent FPCFactory plugin types, viz. ChannelFactories, IORequestFactories and MapperFactories, which are responsible for the creation of new Channels, IORequests and Mappers respectively. All implementations of a single plugin type are tasked with creating the same class of FPC modules, but are allowed different behaviours; for example, the HTTPChannelFactory and the TinyOSChannelFactory both create Channel objects, but the first are used to connect with RESTful web APIs, whereas the second are interfaces to TinyOS networks.

The introduction of a new modular FPCFactory Plugin System resulted in a complete redesign of the Device Descriptor itself, which changed its layout to accommodate a modular structure that closely follows the new FPCFactory architecture. The new Device Descriptor is composed of the following elements:

- **Preamble:** Represented by the root <device> tag, this section contains a textual description of the endpoint, and a list of XML namespaces. As shown later, namespaces are employed to select the various Device Descriptor features and FPCFactory plugins required for the creation of an FPC object;
- **Attribute declarations:** A list of all the Attributes exposed by the device. It must be enclosed in an <attribute> XML tag;
- **Channel declarations:** This section contains the configuration options of all Channel objects required to communicate with the remote device. Its contents are parsed and interpreted by the individual ChannelFactory plugins, whose structure is discussed in section 4.1;
- **Request declarations:** This part of the Device Descriptor is reserved for the declaration of all IORequest objects needed to communicate with a remote device. The elements hereby contained are processed by the IORequestFactory plugins, as per instructions given in section 4.1;
- **Message declarations:** A section reserved for the declaration of all data structures required to exchange information with a node of the network. Its contents are directly interpreted by the MapperFactory plugins to create new Message mappers, as described in section 4.2;
- **Operation declarations:** As shown in section 4.4, this final portion of

the Device Descriptor contains all PerLa `Scripts` employed to control the remote device.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<device type="test" xmlns="http://perla.dei.org/device">
  <attribute>
    <!-- Attribute declarations -->
  </attribute>
  <channel>
    <!-- Channel declarations -->
  </channel>
  <message>
    <!-- Message declarations -->
  </message>
  <request>
    <!-- IORequest declarations -->
  </request>
  <operation>
    <!-- Operation and Scripts -->
  </operation>
</device>
```

Listing 3.1: The skeleton of the new XML Device Descriptor.

As briefly explained in previous paragraphs, XML namespaces constitute a fundamental element of the `FPCFactory` Plugin System, since they are used to select which factory component must be employed for the creation of a specific `FPC` module. A practical example of this concept is available in figure 3.4: this Device Descriptor excerpt specifies two plugin namespaces, one for a `HTTPChannel` (`http://perla.dei.org/fpc/channel/http`), and the other for a `JSONMapper` (`http://perla.dei.org/fpc/message/json`); these two values, once processed

by the /texttttFPCFactory, are used for selecting the specific plugin that will be used for constructing the corresponding `FPC` components. It is important to note that the `http://perla.dei.org/device` and `http://perla.dei.org/device/instructions` namespaces included in this example are mandatory, as they respectively define the Device Descriptor base elements and all available `Script` instructions.

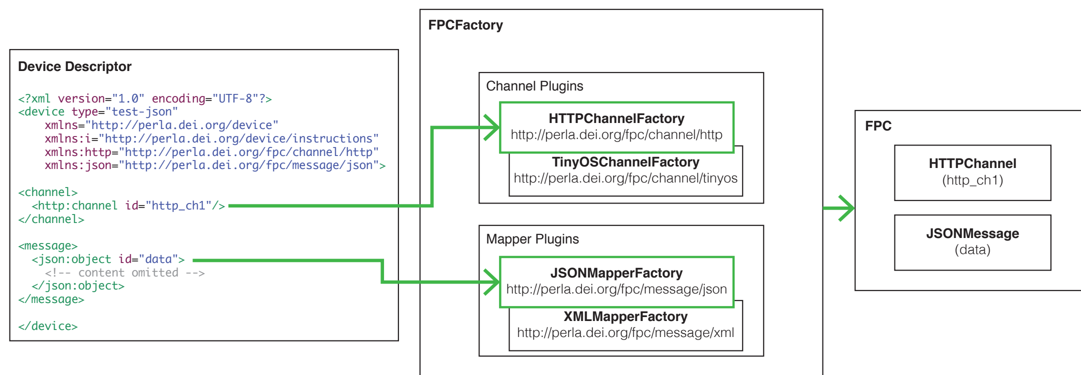

Figure 3.4: Namespace-guided FPC creation.

Every `FPCFactory` plugin is also responsible for defining the particular XML syntax to be used in its Device Descriptor section. This feature endows plugin authors with the opportunity to specify a custom set of options to be used for the creation of their `FPC` modules. The possibility of using a custom syntax is key to the new `FPCFactory` Plugin System, since different types of plugins may require totally different configuration values in order to be used or even initialized (e.g., communicating over a HTTP `Channel` is considerably different than sending data on a low-power mesh network).

3.4 Asynchronous interaction paradigm

The last differences between the New and Classic Middleware architectures lies in the technique employed to interconnect internal modules of the PerLa software infrastructure. The New Middleware introduces a fully asynchronous interaction

paradigm based on non-blocking method invocations and event-driven programming techniques, which deviates profoundly from the mechanism previously promoted in the Classic design.

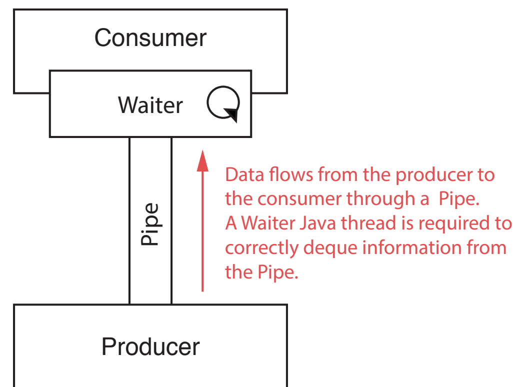

Figure 3.5: A typical `Pipe-Waiter` connection in the Classic PerLa Middleware

Within the previous Middleware architecture, a connection between two different modules was achieved by means of a decoupling element dubbed `Pipe`, a one-way message queue designed to shuttle data elements from a software component to its intended receiver. This system proved to be crucial in the first development stages of PerLa, as its generic interface allowed the early designers to experiment with several competing architectures and component combinations. However, its flexibility came at a cost, both in terms of performances and API readability. First of all, each `Pipe` allocated an initial memory cache of 10 elements. Moreover, the receiving end of a `Pipe`, namely the `Waiter`, was required to instantiate a Java thread dedicated solely to the reception of data messages. The widespread use of the `Pipe-Waiter` paradigm thus led to the proliferation of threads and to an overuse of memory, which negatively impacted the overall system efficiency. In addition, the loosely coupled interaction paradigm promoted by the Classic Middleware resulted in a weak API that lacked intent and semantic clarity.

The asynchronous, event-driven architecture implemented in the New Middleware overcomes all aforementioned drawbacks, and improves system performances in

terms of both throughput and scalability. Differently from the deprecated `Pipe`-driven system, this new design fosters a direct exchange of information between data producers and data consumers; information is no more delivered using a mandatory middleman (i.e., the `Pipe`), but is explicitly handed over to the intended recipients. This interaction paradigm is based on the *Hollywood Principle*, a software design methodology whose tenets are summarized by the motto "*don't call us, we call you*", that encourages the development of highly-cohesive, low-coupling APIs,

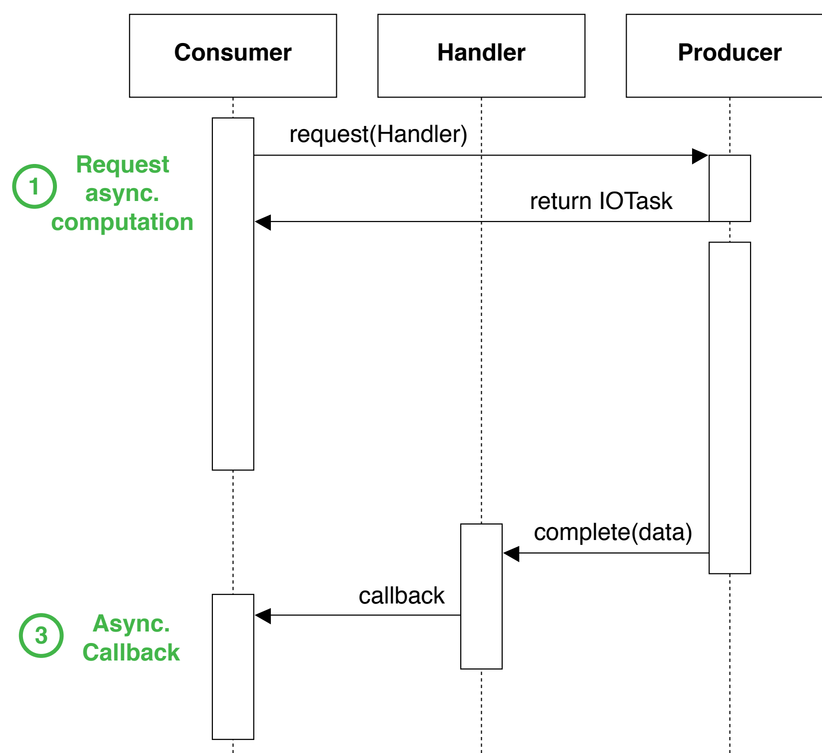

Figure 3.6: Sequence diagram of an asynchronous method call. Note that the consumer and the producer continue their execution in parallel.

Every asynchronous method call in the New PerLa Middleware is identified by the following characteristics:

- **Does not block:** Calls to an asynchronous method never block; control of the execution flow is immediately returned to the caller, and the requested

computation is executed asynchronously. This characteristics reduces the number of Java threads that the caller module needs to instantiate;

- **Returns a Task object:** Asynchronous method calls do not return the immediate result of a computation. Instead, they return a `Task`, i.e., an object that can be employed to stop the ongoing operation or to query its current state of progress;
- **Defers the delivery of results:** The effective result of an asynchronous method is notified through a `Handler` function, which is invoked as soon as the computation terminates.

For an in-depth description of the actual asynchronous APIs implemented within the PerLa Middleware, refer to chapter 4.

Chapter 4

In-depth component description

4.1 Communicating with Channels

`Channel` is an interface for performing I/O operations. It represents the principal abstraction used by the middleware to communicate with hardware devices and external software services.

The `Channel` interface is not tied to any specific technology or communication stack; as a result of this design choice, a wide variety of data management tasks, including but not limited to, networking, file handling, and automatic data generation can be implemented as `Channel`s. This concept is taken even further by the `SimulatorChannel`, a particular `Channel` implementation that generates random data samples that allow the PerLa Middleware to be tested even when no devices are connected to it.

The current Middleware architecture encourages the creation of several highly specialized `Channel`s, which are usually developed around third-party communication libraries. `HTTPChannel`, a `Channel` providing support for HTTP communications, is an excellent example of the advantages of this design strategy. Implemented as a simple wrapper around Apache's HTTP Components toolkit, its development only required a basic understanding of the HTTP protocol; yet `HTTPChannel` is a

```
1  public interface Channel {
2
3          public String getId();
4
5          public IOTask submit(IORequest request, IOHandler handler)
6                          throws ChannelException;
7
8          public void setAsyncIOHandler(IOHandler handler)
9                          throws IllegalStateException;
10
11         public boolean isClosed();
12
13         public void close();
14
15 }
```

Listing 4.1: The Channel interface

fully compliant HTTP/1.1 client.

Upon instantiation, `Channel`s are open and ready to be used. They may be optionally closed to relinquish unused resources by invoking the `close()` method. Once closed, a `Channel` cannot be re-opened, and every subsequent attempt to perform an I/O operation will fail causing a `ChannelException` to be thrown. The current state of a `Channel` can be probed through its `isClosed()` method.

Bytes sent or received with a `Channel` are encapsulated in a `Payload` object. As shown in listing 4.2, the `Payload` interface allows all Middleware components to handle different data types with a common set of methods, regardless of their individual encoding. `Payloads` will be the subject of further discussion in section 4.2

All user-initiated I/O operations begin with an invocation of the `Channel.submit()` method. As can be seen in listing 4.1, `submit()` is a direct implementation of the asynchronous interaction paradigm introduced in section 3.4. The emphasis on asynchronous execution is underscored by the absence of blocking operations in the `Channel` interface. This aspect is of paramount importance for the entire Middleware design, as implementing a truly asynchronous system would prove

```java
public interface Payload {

	public Charset getCharset();

	public InputStream asInputStream();

	public ByteBuffer asByteBuffer();

	public String asString();

}
```

Listing 4.2: The Payload interface

impossible if such feature were not provided by its core data access layer.

4.1.1 Instantiating new Channels

`Channel`s are created by means of the `ChannelFactory` interface, a reification of the Factory design pattern that allows polymorphic instantiation of new object classes.

By using a Factory instantiation model, the choice of a particular `Channel` implementation can be postponed from compile time to run time. This technique allows the Middleware to dynamically adapt in response to environment changes, and to support extension through the addition of new user-defined `Channel`s. For further information regarding the Factory pattern and its other uses inside the PerLa Middleware, refer to section 3.3.

All the information required to create a new `Channel` is stored inside a `ChannelDescriptor`. As shown in listing 4.3, this configuration object is the only parameter required to correctly invoke the `createChannel()` method.

Each `ChannelFactory` is tied to a specific communication technology; therefore, it can only accept a single class of `ChannelDescriptor` objects. For example, the `HTTPChannelFactory` parses `HTTPChannelDescriptor`s and creates `HTTPChan-`

```
public interface ChannelFactory {

        public Class<? extends ChannelDescriptor>
                        acceptedChannelDescriptorClass();

        public Channel createChannel(ChannelDescriptor descriptor)
                        throws InvalidDeviceDescriptorException;

}
```

Listing 4.3: The ChannelFactory interface

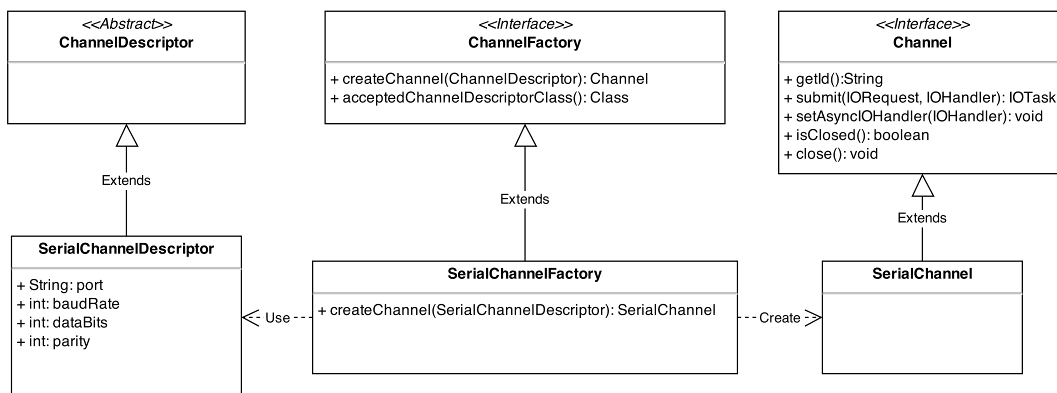

Figure 4.1: Class diagram of the Channel layer

nels, whereas an hypothetical `SerialChannelFactory` would consume `SerialChannelDescriptors` to create `SerialChannels`. Failure to provide a suitable `ChannelDescriptor` object will cause the `createChannel()` method to throw an `InvalidDeviceDescriptorException`.

The `acceptedChannelDescriptorClass()` method can be used to dynamically discover which `ChannelDescriptor` type is supported by a specific `ChannelFactory`. This method is the fulcrum of the `Channel` Plugin System, as it allows the Middleware to invoke the most appropriate `ChannelFactory` using only information available at runtime.

`ChannelDescriptor` objects are automatically created by the Middleware using the information contained in the Device Descriptor XML files. This binding process

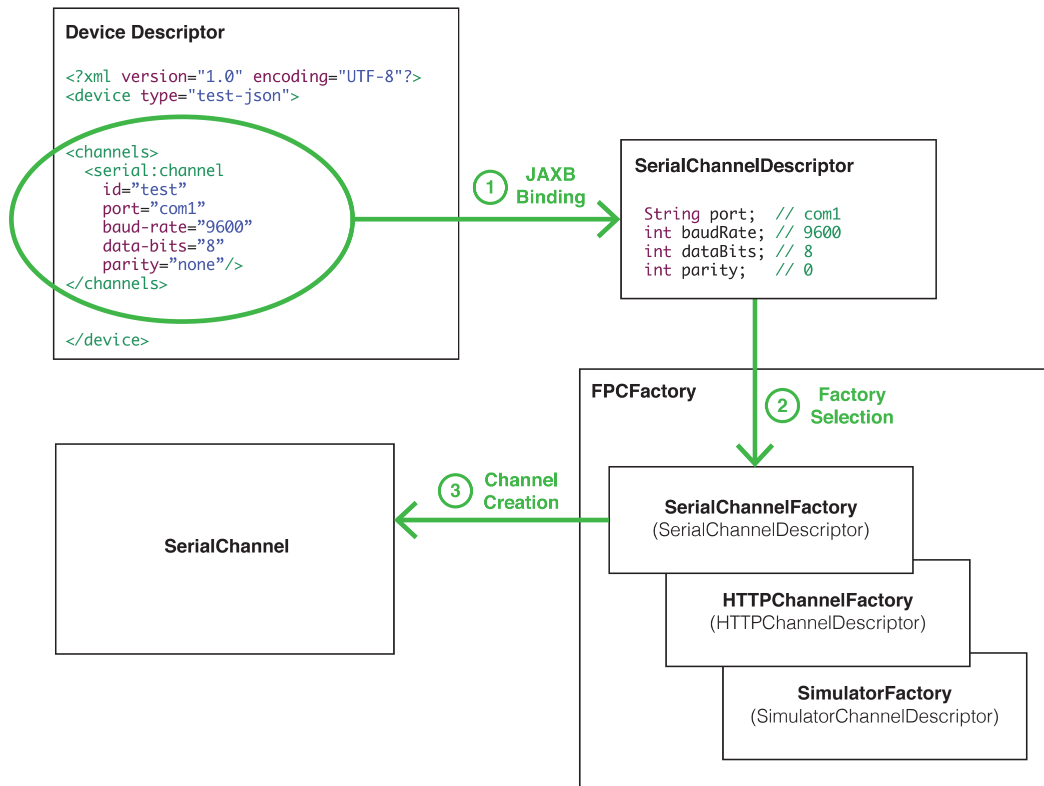

Figure 4.2: The Channel creation process

This figure illustrates the `Channel` creation process executed by the Middleware upon reception of a new Device Descriptor.

1. JAXB binds the XML Device Descriptor to an appropriate `ChannelDescriptor` object using namespace information
2. A suitable `ChannelFactory` is selected at runtime using the `acceptedChannelDescriptor-Class()` method
3. The information contained in the `SerialChannelDescriptor` is used to create a new `Seri-alChannel`

---

is performed by the JAXB library, which is also responsible for instantiating the correct `ChannelDescriptor` class using XML Namespace information. Figure 4.2 illustrates this technique, and ties it together with the other operations described in this section.

It is important to note that a single JVM instance running the PerLa Middleware

```
public interface IORequest {

        public String getId();

        public void setParameter(String name, Payload payload);

}
```

Listing 4.4: The IORequest interface

may host several Channel objects of the same type, at the same time. Several devices can use the same communication technology, and the ChannelFactory may determine that it's best to create an individual Channel for each one of them. This behaviour is fostered by the new ChannelFactory architecture, and is considered idiomatic design; hence, it would not be uncommon to implement the hypothetical SerialChannelFactory introduced in the previous paragraphs so that every serial port is handled by a different SerialChannel instance.

4.1.2 IORequest management

IORequest is the base object interface employed to interact with a sensing node connected to the Middleware. It contains two types of information: the payload to be transferred, and Channel-dependent data needed for a correct communication with the endpoint device.

Every Channel implementation is bundled with its own custom IORequest class. Following up on previous examples, the HTTPChannel package contains a HTTPIORequest object, whereas the fictitious SerialChannel would be supplied with a SerialIORequest class of request objects. This additional level of indirection is necessary since different communication technologies require different settings to establish end-to-end connectivity; therefore, a universal IORequest object would soon prove to be a limiting factor for the extension of the Middleware.

As shown in listing 4.4, payload data can be set in an IORequest by means of the

`setParameter()` method. `Payloads` are addressed by name, and a single `IORequest` implementation may support several at once. The exact set of `Payload` parameters accepted by an `IORequest` class depends on the design of its matching `Channel`; for example, the `HTTPChannel` implementation supports three: an 'entity' payload (request body), a 'query' payload (an URL-encoded string), and a 'path' payload (a path component used to identify a single resource accessible from the base URL).

`IORequests` are disposable objects; they are created, submitted to a `Channel`, and garbage collected once the communication is over. Creation is performed by means of a factory interface dubbed `IORequestBuilder`, which allows the Middleware to build new copies of an `IORequest` from a fixed template. It is important to note that request objects built using this technique do not contain `Payload` parameters; these are to be added manually before submitting the `IORequest` to a `Channel`.

Besides `IORequest` creation, the `IORequestBuilder` interface can be used to dynamically discover which `Payload` parameters are supported by an `IORequest`. This functionality, exposed through the `getParameterList()` method, is a crucial component of the Middleware Plugin System, as it allows *Script* instructions to determine whether an `IORequest` was populated with all the necessary `Payload` parameters or not. This concept will be the subject of further analysis in section 4.3.

A single device connected to the PerLa Middleware is generally managed using several `IORequestBuilders`, any one of which is responsible for creating a request object suitable to control a single aspect of the interaction with the endpoint. The main advantage brought by this templating mechanism is that `Channel`-related configuration settings are only specified once, hence the same `IORequest` structure can be reused multiple times to transport different payload information.

```
1 public interface IORequestBuilder {
```

```
2
3       public String getRequestId();
4
5       public IORequest create();
6
7       public abstract List<IORequestParameter> getParameterList();
8
9       public static class IORequestParameter {
10
11              public String getName() {
12                      return name;
13              }
14
15              public boolean isMandatory() {
16                      return mandatory;
17              }
18
19      }
20 }
```

Listing 4.5: The IORequestBuilder interface

REST APIs are an excellent use case to demonstrate the aforementioned concept, as every operation on a RESTful resource can be easily abstracted using an appropriately configured request builder. By using `HTTPIORequestBuilder` objects, HTTP protocol information (base URL, method, header, ...) are specified one single time only for every REST endpoint. Once this step is done, the API can be invoked just by building new `IORequest`s and submitting them to a `HTTPChannel`.

`IORequestBuilder`s are created by means of an `IORequestBuilderFactory`, an object that implements the now familiar Factory design pattern. Creation proceeds as follows: the request template is loaded from an XML Device Descriptor, bound to an appropriate `IORequestDescriptor`, and processed by the `IORequest-BuilderFactory` to create the corresponding `IORequestBuilder`. Similarly to

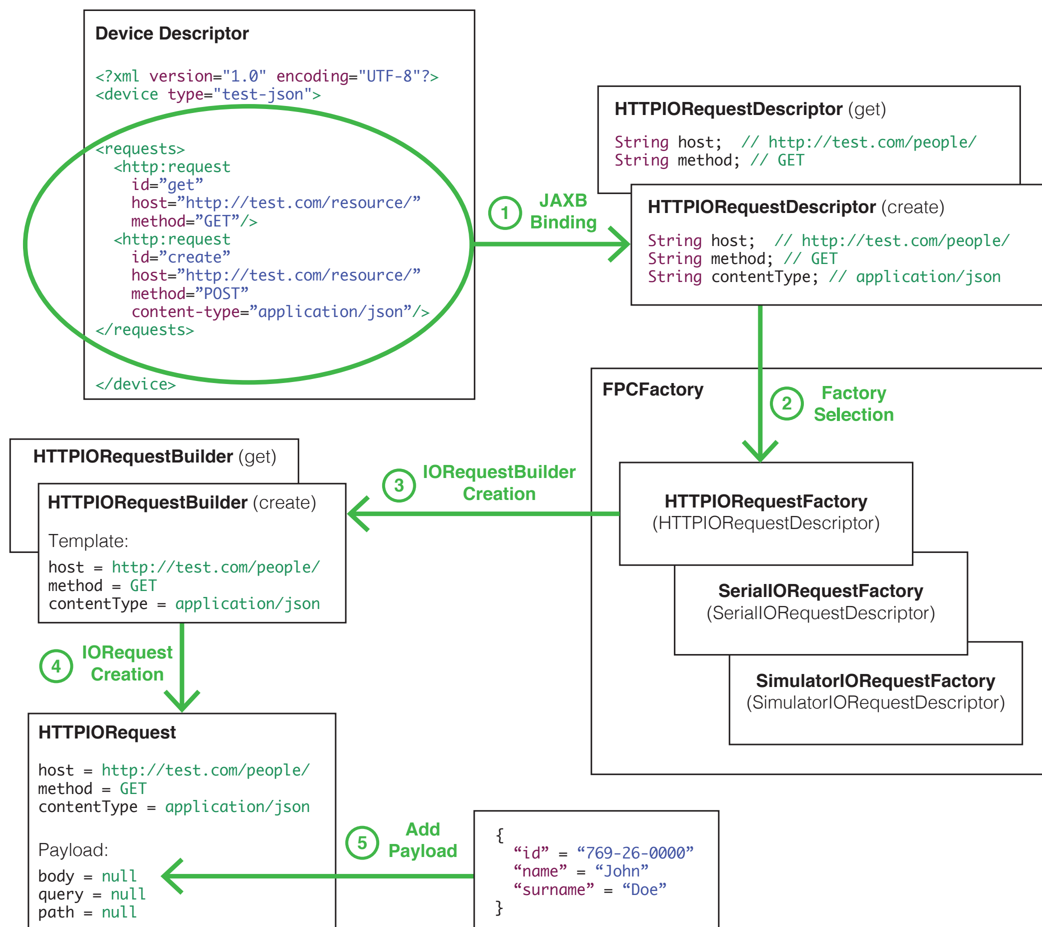

Figure 4.3: The IORequest creation process

This figure illustrates the autonomous creation of `IORequest` objects. Steps 1 to 3 are performed only once after receiving the Device Descriptor, whereas steps 4 and 5 are repeated every time the REST API is to be invoked.

1. JAXB binds the XML Device Descriptor to an appropriate `IORequestDescriptor` object using namespace information
2. A suitable `IORequestBuilderFactory` is selected at runtime using the `acceptedIORequestDescriptorClass()` method
3. The information contained in the `IORequestDescriptor` is used to create a new `IORequestBuilder`
4. The `IORequestBuilder` is used to create new `IORequest` copies using the internal template
5. The newly created `IORequest` objects can be populated with `Payload` parameters as needed

what already seen in the previous section, every `IORequestBuilderFactory` implements an `acceptedIORequestDescriptorClass()` method, which can be used to dynamically determine if a factory object can parse a specific type of `IORequestDescriptor`. It should come as no surprise that every `IORequestBuilder` class is provided with complementary `IORequestBuilderFactory` and `IORequestDescriptor` implementations.

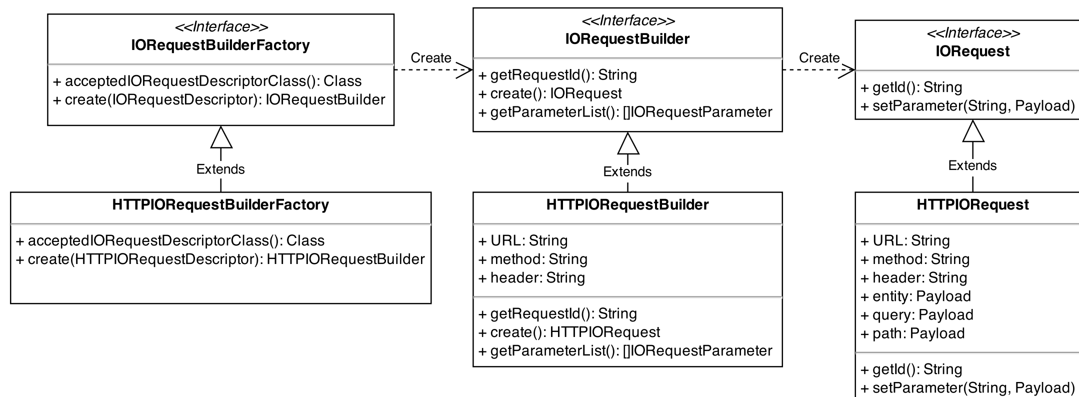

Figure 4.4: The extended IORequest class diagram. For additional information about the IORequestParameter object consult listing 1.5.

4.1.3 Handling asynchronous I/O operations

As mentioned in previous sections, communication with a device connected to the PerLa Middleware is achieved by means of the `Channel.submit()` method. Invocations of `submit()` are non-blocking; control flow is immediately returned to the caller, thus allowing other computations to be performed while the requested I/O operation is being processed.

As can be seen in listing 4.1, `submit()` requires two parameters: an `IORequest` and an `IOHandler` callback object. The former specifies which I/O operation is to be performed, while the latter allows the caller to be asynchronously notified of its completion.

The `IOHandler` interface is composed of two methods, namely `complete()` and

```
public interface IOHandler {
        public void complete(IORequest request, Optional<Payload>
            result);

        public void error(IORequest request, Throwable cause);
}
```

Listing 4.6: The IOHandler interface

```
public interface IOTask {
        public void cancel();

        public IORequest getRequest();

        public boolean isCancelled();

        public boolean isDone();
}
```

Listing 4.7: The IOTask interface

`error()`, which are invoked when processing of an I/O operation comes to an end. It is important to note that both these methods always carry context information in the form of an `IORequest`, which is guaranteed to be the same exact object used for starting the I/O operation whose completion is being notified. For this reason, `IOHandler` can be considered the nexus of the asynchronous invocation model, as it connects `IORequest` objects with the outcome of the corresponding I/O operation performed by the `Channel`.

Semantically, an invocation of the `complete()` method is always associated with the successful termination of an I/O operation. As shown in listing 4.6, this method includes an optional `Payload` object, that contains all data received from the endpoint device. A call to `complete()` with an empty `Payload` indicates that the I/O operation was completed without errors, but no data was received. Conversely, an invocation of the `error()` method indicates that the I/O operation was aborted before completion. In this case the cause of failure is always notified through the `cause` parameter.

From the point of view the Java memory model, the `Channel.submit()` creates a happens-before relationship with `IOHandler.complete()` and `IOHandler.error()`, viz. any side effect generated by the code that led to the `submit()` invocation is guaranteed to be visible in the `complete()` and `error()` callback methods.

Asynchronous execution does not imply loss of control; ongoing I/O operations can be monitored or cancelled by means of the `IOTask` object acquired upon submitting an `IORequest`. Listing 4.7 shows all methods of the `IOTask` interface; method names are self explanatory, and the reader should be able to deduce their purpose just by analyzing their signature. The only nuance worth mentioning is that `isCancelled()` always implies `isDone()` (i.e., all cancelled I/O operations are also complete), while the opposite does not hold (i.e., not all complete I/O operations were cancelled).

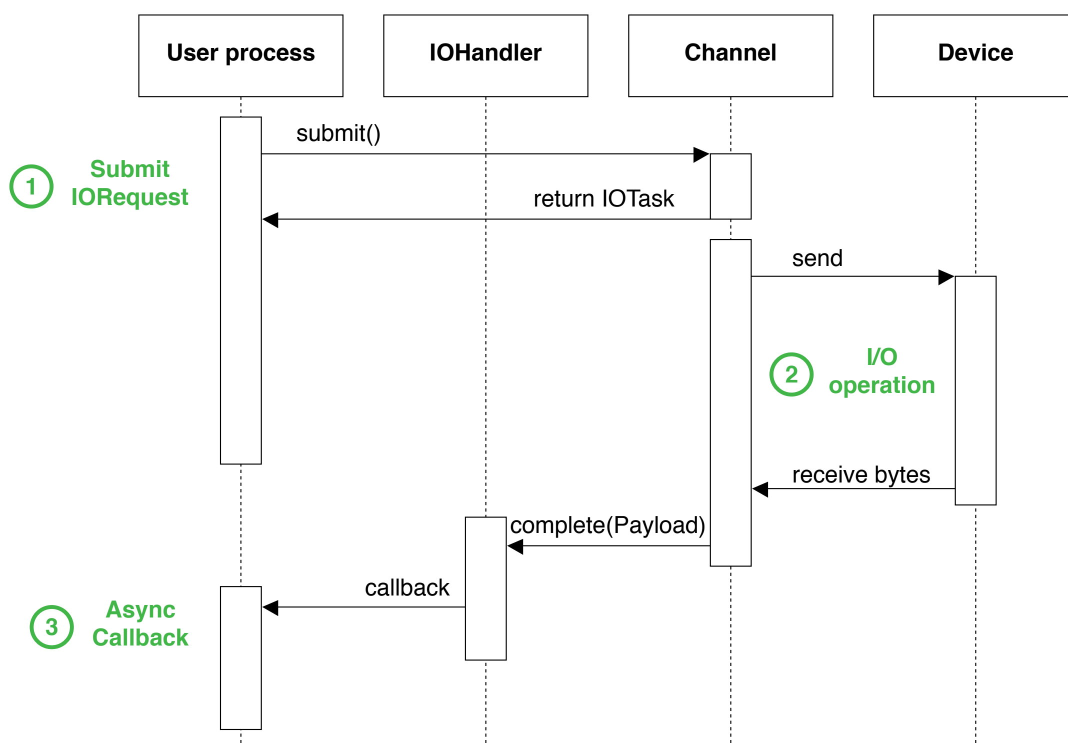

Figure 4.5: Sequence diagram of an asynchronous I/O operation. Note that the user process and the I/O operation are executed in parallel.

The `Channel` interface is also designed to manage completely asynchronous I/O operations, namely communication efforts spontaneously initiated by the remote device. This communication model is popular among WSNs, as it is often employed to handle periodic data streams or events happening at irregular intervals. Such I/O operations can be handled through a catch-all `IOHandler` set with the `setAsyncIOHandler()` method (listing 4.1). Since the communication is not initiated by the Middleware, the `complete()` and `error()` callback methods will be invoked with the `IORequest` parameter set to `null`.

4.2 Handling data

`Payload` is a container for raw sequences of bytes. In spite of its simplicity, this class forms the foundation of the entire PerLa Middleware, as it is the vessel that conveys all information passing through the `Channel` interface.

The data encapsulated in a `Payload` object is accessed one byte at a time; this granularity level is ideal for the implementation of an I/O access layer, whose sole concern consists in the transmission of information between two endpoints, but is not suited to other forms of data management. Processing the information contained in a `Payload` can be unwieldy and unnecessarily complex; the byte-oriented interface doesn't provide any facility for leveraging the underlying structure of the enclosed data, and even a simple action like retrieving a value in a complex data structure can easily become a daunting task.

4.2.1 The Message interface

`Message`s are structured data containers that enclose a group of individual items called fields. The chief advantage that this data structure provides over the simpler `Payload` object consists in the possibility of addressing information by field name, a convenient feature that dispenses with the burden of managing data in byte-sized chunks. The methods available in the `Message` interface are shown in listing 4.8.

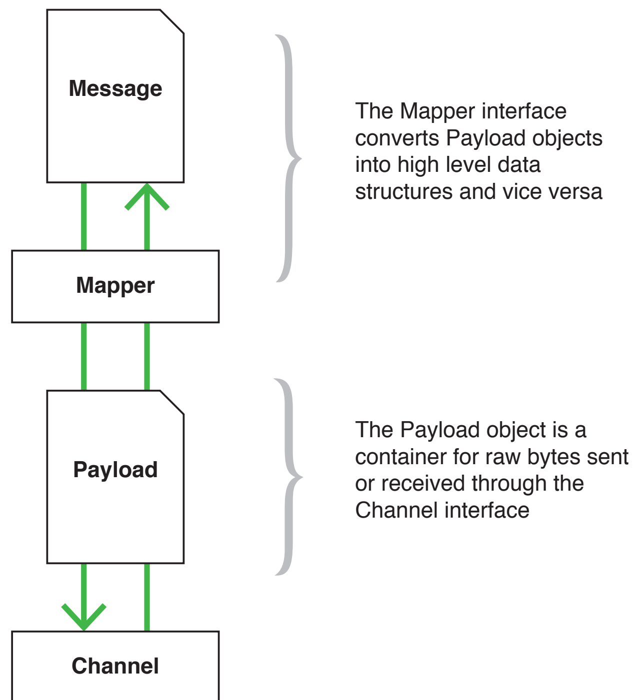

Figure 4.6: Relationship between the `Channel`, `Payload`, `Mapper` and `Message` objects.

```
public interface Message {
    public String getType();

    public boolean hasField(String name);

    public Object getField(String name)
                throws IllegalArgumentException;

    public void setField(String name, Object value)
                throws IllegalArgumentException;

    public void appendElement(String name,Object element)
                throws IllegalArgumentException;
```

```
14
15     public boolean validate();
16 }
```

Listing 4.8: The Message interface

The specific structure of a `Message` is defined by its type, which can be queried through the `getType()` method. This property unequivocally identifies the set of fields contained in a `Message` in terms of field **name**, field **type** and field **qualifier**.

The field **name** is a textual attribute that uniquely identifies one specific data item in the scope of a single `Message`. It can be used to retrieve or set the value of a field through the `getField()` and `setField()` methods respectively.

The **type** attribute defines the set of legal values that can be stored in a field, together with the operations that are allowed on those values. It is worth mentioning that this information is used to statically verify the type safety of nearly all data management operations performed on a `Message` (consult section 4.3 for additional information). The PerLa Middleware currently supports six primitive types:

- **INTEGER**: a 32 bit signed two's complement integral data type
- **FLOAT**: a single-precision 32 bit IEEE 745 floating point
- **BOOLEAN**: a type with only two values, true or false
- **STRING**: a string of characters with UTF-16 encoding
- **TIMESTAMP**: a date with timezone, currently implemented using Java's `ZonedDateTime` class.
- **ID**: a unique label that identifies a single node connected in a PerLa managed network. The current implementation uses a 32 bit integer.

Besides the data types presented above, fields can also be configured to hold nested `Messages`. In this case, the type attribute must be set to the particular type of `Message` that is to be stored in the field.

The **qualifier** attribute is employed to define additional field properties. It can

be set to one of the following values:

- **SIMPLE**: a normal field whose value can be altered and retrieved using the `setField()` and `getField()` methods respectively.
- **LIST**: a field that can hold multiple elements of the same type. New values can be added with the `appendElement()` method, and the entire list can be retrieved through the conventional `getField()` method. List-qualified fields preserve the order of insertion of the individual elements.
- **STATIC**: a field whose value is statically set when the `Message` type is declared. Any attempt to modify a statically-qualified field with either the `setField()` or the `appendElement()` methods will cause an exception to be thrown. It is important to note that static field values are set on a per-type basis; this means that all `Message`s of the same type will share the same field values for each static field (if any).

4.2.2 Working with Messages: the Mapper interface

`Message` objects are managed by the `Mapper` component. Its interface, available in listing 4.9, groups all the functionalities needed to handle a specific variety of structured information. The one-to-one relationship between `Mapper`s and data types is epitomized by the `getMessageType()` method, whose return value indicates which `Message` class is supported by a particular `Mapper`. This method is extensively employed by the Middleware to sift through a collection of `Mapper`s, in order to find one that is best suited for handling the information currently being processed.

```
public interface Mapper {

    public String getMessageType();

```

```
5       public FieldDescriptor getFieldDescriptor(String name);
6
7       public Collection<FieldDescriptor> getFieldDescriptors();
8
9       public FpcMessage createMessage();
10
11      public FpcMessage unmarshal(Payload payload);
12
13      public Payload marshal(FpcMessage message);
14
15  }
```

Listing 4.9: The Mapper interface

Interactions with a `Mapper` usually begin with a call to the `createMessage()` method, whose execution results in the creation of an empty `Message` instance. Despite its unsurprising outcome, this method draws once again our attention to the close relationship between `Mapper`s and data types. Every `Mapper` instance is in fact committed to the management of a precise class of information, hence all `Message`s created with the `createMessage()` method will share the same data type property, and, consequently, the same set of fields. The interdependence between a `Mapper` and its assigned type is accentuated even further by the `getFieldDescriptor()` and `getFieldDescriptors()` methods, which can be used to analyze the internal field structure characterizing all `Message` objects that the `Mapper` creates. This introspective capability is extensively exploited in the Execution Engine to check whether a *Script* is type-safe or not (see section 4.3 for further details).

As explained in the introductory paragraphs of this section, `Message` objects are a convenience introduced for simplifying data management operations in the PerLa Middleware. They provide structured access to information, a familiar set of primitive data types, and a selection of tools for combining basic values into complex data structures. In spite of these advantages, the `Message` interface is a high level

abstraction that cannot be employed where a `Payload` is expected, since its contents are not directly accessible as a simple sequence of bytes; as a consequence, `Message`s can't be used for any kind of I/O operation. This structural gap is bridged by the `marshal()` and `unmarshal()` methods of the `Mapper` interface. As can be seen by analyzing their respective signatures, these two methods can be used to convert `Message` objects into `Payload`s and vice-versa. This additional `Mapper` functionality brings to light yet another aspect of the PerLa data management layer, namely its ability to work with different representations of binary data.

Every `Mapper` is in fact created to support a single data format; `JSONMapper` instances, for example, handle JSON-formatted byte streams, whereas `URLEncodedMapper`s specialize in the conversion of URL-encoded HTTP entities. The structure of the `Message`s created by a `Mapper` and the data format they can be marshalled unto are not orthogonal concerns, as the choice of a specific binary representation may prevent the use of some of the previously discussed field attributes. The URL-encoded format, for example, is defined as a flat collection of key-value pairs, with no support for nested data structures; hence, the corresponding `URLEncodedMapper` class could never be used to create and manage `Message`s with nested fields. The close connection between a `Message` and its corresponding binary format manifests itself in the design of the `Mapper` component, specifically in the decision to coalesce the marshalling/unmarshalling mechanism, and the more general `Message` management methods (`createMessage()`, `getFieldDescriptors()`), under the same interface. The specific methodology for creating `Mapper` objects, and for defining their distinctive data format and `Message` type, will be subject of additional discussion in the remainder of this chapter.

Before this section comes to an end, it is worth putting into context the role occupied by the `Mapper` inside the PerLa Middleware. The additional decoupling provided by the `Mapper` builds over the pluggable `Channel` interface, thus allowing the payload format to be selected independently of the I/O stack. This is an important characteristic of the Middleware design, as even the simplest communi-

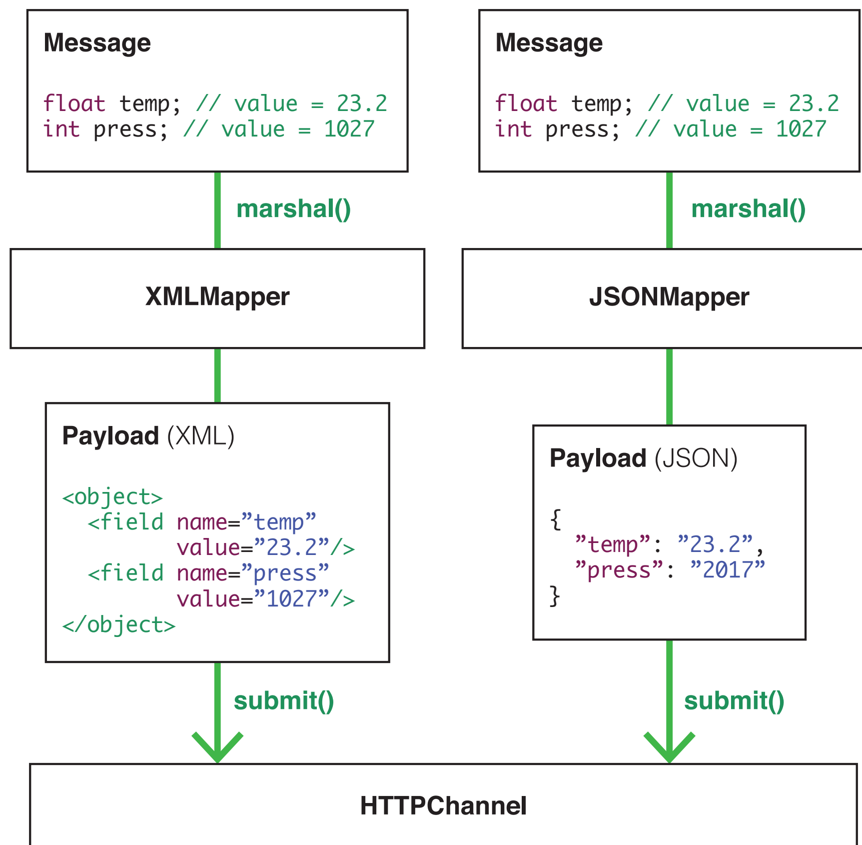

Figure 4.7: Using a single `Channel` to transmit data marshalled with different `Mappers`

cation protocol usually requires several `Message` structures, viz. several `Mappers`, for exchanging data between two endpoints.

4.2.3 Creating Mappers and defining Message structures

New `Mapper` objects are created by means of the `MapperFactory` interface.

Its design follows the same concepts explained in previous sections; the `acceptedMessageDescriptorClass()` method returns the type of `MessageDescriptor` objects that can be used with the `MapperFactory`, while the `createMapper()` method consumes a `MessageDescriptor` to create a `Mapper`. However, differently from all factory components described so far, the creation of a new object calls

```
1  public interface MapperFactory {
2
3     public Class<? extends MessageDescriptor>
4        acceptedMessageDescriptorClass();
5
6     public Mapper createMapper(MessageDescriptor descriptor,
7        Map<String, Mapper> mapperMap, ClassPool classPool)
8                 throws InvalidDeviceDescriptorException;
9
10 }
```

Listing 4.10: The Mapper Factory interface

for two additional parameters other than the descriptor itself: a `ClassPool`, and a map of `Mappers`. These extra items contain a reference to previously built `Mappers`, and can be used to check whether nested `Message` fields are properly declared or not.

The `MapperFactory` interface is an additional extension point available to PerLa users, and can be leveraged to introduce support for new binary formats and information encoding schemes. As a consequence, every installation of the Middleware will contain a wide variety of `MapperFactory` implementations, each of which is dedicated to a single data format. Instances of the previously introduced JSON and URL-Encoded mappers, for example, are created by two distinct `MapperFactory` objects, namely `JSONMapperFactory` and `URLEncodedMapperFactory`. This design is a substantial improvement on the previous middleware architecture, as it ensures that every `MapperFactory` object is responsible for managing the quirks of only a single data format.

Moreover, every `MapperFactory` implementation is bundled with a custom `MessageDescriptor` object, whose class name is exposed by the aforementioned `acceptedMessageDescriptorClass()` method. The additional complexity deriving from this design choice is more than made up for in type safety and flexibility, as each different `MessageDescriptor` may be implemented to closely represent the idiosyncratic characteristics of its corresponding data format. A concrete example

of this concept comes from the `URLEncodedMessageDescriptor` class, which prevents the creation of `Message`s that don't comply with the URL-encoded format by disallowing non-primitive fields. Having multiple `MessageDescriptor`s also means that different `Message`s are not forced to abide by the same set of rules; the limits imposed on URL-encoded messages are not universal, and in fact the `JSONMessageDescriptor` refrains from applying them. Furthermore, `MessageDescriptor` objects can adopt a custom lexicon for expressing the PerLa-specific concepts of *message*, *field* and *field type*. Take for example listings 4.11 and 4.12. These two XML snippets show how the vocabulary employed in message declarations varies with the data format (field are dubbed *member* in JSON, and *parameter* in URL-Encoded strings). Using a terminology that best suits the actual data format makes `Message` declarations idiomatic, reminiscent of the corresponding real-world objects and therefore easier to use.

4.2.4 Managing multiple message types

It is not uncommon for a single device to communicate using multiple message formats; developers may choose to encapsulate different information inside different data structures, which the receiver must correctly identify to decipher their contents. In such cases, every `Message` exchanged between the two endpoints is tagged with a data type value, i.e., a common field that advertises the type of information being transferred. This technique is widespread among firmware developers, since it can be easily implemented with most programming languages (C/C++ support it by design through `tagged unions`).

The PerLa Middleware implements various techniques to cope with sensor nodes that communicate using multiple message formats. First of all, only the data structures that can actually be received are considered when unmarshalling a byte stream; if under the current conditions a device only sends a subset of its available message types, then the Middleware can immediately rule out the unmatching ones. As it will be discussed later, a collection of expected data formats is automatically curated by the `FPC` by cross-comparing information excerpted from

```
1
2  <js:object id="coord">
3     <js:member name="lon" type="string"/>
4     <js:member name="lat" type="string"/>
5  </js:object>
6
7  <js:object id="main">
8         <js:member name="temp" type="float"/>
9         <js:member name="pressure" type="float"/>
10        <js:member name="humidity" type="float"/>
11        <js:member name="temp_min" type="float"/>
12        <js:member name="temp_max" type="float"/>
13 </js:object>
14
15 <js:object id="wind">
16        <js:member name="speed" type="float"/>
17        <js:member name="deg" type="float"/>
18 </js:object>
19
20 <js:object id="weather">
21        <js:member name="coord" type="coord"/>
22        <js:member name="main" type="main"/>
23        <js:member name="wind" type="wind"/>
24 </js:object>
```

Listing 4.11: A compound JSON message declared using the JSONMessageDescriptor (XML notation). Note that the data type of all fields inside `weather` message is a reference to a previously declared `Message`.

```
1
2 <ue:message id="urlencoded_message">
3         <ue:parameter name="temperature" type="float"/>
4         <ue:parameter name="pressure" type="float"/>
5         <ue:parameter name="location" type="string"/>
6         <ue:parameter name="key" qualifier="static" type="integer"
             value="5"/>
7         <ue:parameter name="timestamp" type="timestamp" format="d MMM
             uuuu HH:mm"/>
8 </ue:message>
```

Listing 4.12: An URLEncoded message declaration. Thanks to the custom `URLEncodedMessageDescriptor`, trying to create a non-primitive field results in an exception. Note the custom `format` attribute on the timestamp field, which is employed to define the encoding format for dates and times

the Device Descriptor with the current device status. It should be clear that this technique alone is not enough to cover all practical use cases, as it falls short as soon as a device starts sending two or more message varieties concurrently; in such scenarios, PerLa needs to search for clues that will help it recognize how the bytes being received are structured. These clues take the form of *static* fields. When faced with an ambiguous situation, the `FPC` will try unmarshal the bytes received into all expected data types. A congruency check will then be performed on the resulting `Messages`: the data can be considered correctly decoded only when all its static fields match the corresponding Device Descriptor declaration. This methodology can be employed to interact with sensor nodes that make use of tagged data structures.

4.3 Data management: Scripts

`Channels`, `Payloads`, `Mappers` and `Messages` are the core components used by PerLa to exchange data with nodes of a Pervasive System. They provide the supporting infrastructure through which information can be serialized, transmitted and faithfully reconstructed at the receiving endpoint. Taken together, these com-

ponents implement an adaptable transport layer, whose features can be tailored around each device connected to the Middleware: combine a `HTTPChannel` with a `JSONMapper` to obtain a network stack for RESTful services; swap the data layer with an `XMLMapper` if the format changes; add a `ZigbeeChannel` and a `StructMapper` to communicate with low-powered devices in a mesh network. In spite of their individual capabilities, all these components are not enough to glean information from a Sensor Network. The interaction with a sensor node requires far more than a transport layer; in fact it can only occur when data transfer operations follow a strict set of rules, i.e., an *application protocol*. `Channel`s, `Mapper`s and `Message`s provide no more than the basic building blocks needed for the interaction, but their use is to be tightly orchestrated before any purposeful exchange of information can take place.

PerLa *Script*s, just referred as *Script*s in the remainder of this document, implement the kind of structural scaffolding required to organize a series of primitive data management operations into a self-contained, reusable procedure. Their purpose in the PerLa Middleware is twofold: first, to issue commands that conform to the specific protocols used in a Pervasive Systems; second, to act as an impedance matcher between the structured information collected from a sensor network and the record-oriented output of an `FPC`. The PerLa scripting language is one of the most distinctive features of the new Middleware design; a procedural programming tool that can be used to complement and enrich the declarative nature of the Device Descriptor. *Script*s improve the reusability of all existing and future Middleware components, as they can be used to adjust the output of a computation before it's used as the input of another one.

An archetypal example of this concept is given in figure 4.8. This *Script* solves a recurring impedance matching problem: a device that stores multiple samples into a common data structure, and sends them with a single transmission in order to conserve battery power. The result of such aggregation can't be coerced into a sequence of records as-is, as more often than not it contains a mixture of both high frequency and low frequency information (and indeed it does in the current

Figure 4.8: Flattening the content of a nested JSON data structure with a PerLa `Script`

example). The PerLa scripting language can be used to unroll the high frequency content, complement it with the information that remains constant for all samples, and output the resulting records one by one; all without having to resort to a bespoken `Mapper` implementation, as it would prove necessary if the former Middleware were used instead. Moreover, this same script can be easily employed to handle similar aggregation patterns with little or none modifications, as it doesn't depend on any particular `Channel`, `Mapper` or `Message` implementation.

PerLa *Scripts* are also used to address minor compatibility problems that may arise when authoring new Device Descriptors; they can wrap an existing Middleware component and adapt its behaviour to handle an unforeseen usage scenario, convert information between different units of measure, alter a `Message` before it is sent to the intended recipient, or compute aggregations. It is important to note that *Scripts* are slower than pure Java code; an excessive usage in data intensive, real-time applications should be carefully avoided, as it may negatively affect the performance of the entire Middleware. End users are therefore invited to thoroughly test and benchmark their Device Descriptors to eliminate any potential bottleneck before deployment.

4.3.1 Anatomy of a PerLa Script

PerLa `Script` is a full-fledged imperative programming language composed of data management instructions, control flow statements and a powerful expression language. Procedures written in PerLa `Script` are processed by the `Script Engine`, a Middleware module that reads, interprets and executes script instructions. Instructions are specified using a proprietary XML syntax, specifically designed to allow PerLa `Scripts` to be directly embedded in a Device Descriptor. Conflicts with potentially similar XML tags and attributes are avoided through the `http://perla.dei.org/device/instructions` namespace, which is commonly associated with the "`<i:`" prefix. PerLa developers are invited to follow this convention, as the use of a different namespace prefix may create confusion among end users and future Device Descriptor maintainers.

Every `Script` instruction is composed of a *name*, a textual property that identifies a specific type of computation, and a list of *parameters*, name-value pairs that customize its runtime behaviour. With the exception of the `submit` instruction, whose unconventional characteristics are described in the remainder of this section, parameters are always defined using the standard XML attribute syntax. PerLa `Scripts` currently support two different types of parameter values: *literals* and *expressions*. Literals are simple textual strings, which are used as-is by the execution engine to express constant concepts like variable names or immutable values. Expressions, on the other hand, are combinations of variables, operators, functions and constants, whose evaluation produces a new result value. Differently from literals, expressions are prefixed by the $ sign and enclosed in curly braces (`${ ... }`); this cue is employed by the execution engine to determine whether an instruction parameter has to be pre-processed or not prior to being used. Expressions can be used to perform the following actions:

- Arithmetic operations (sum, subtraction, product, division, modulo)
- Logical operations (or, and, not)
- Comparisons (`<, >, !=, <=, >=`)
- Access `Message` fields (dot operator). Multiple dot operators may be applied in succession to access a specific value buried inside a complex data structure (e.g., the expression `${result.environment.temperature}` is used to traverse 3 nesting levels).
- Retrieve `Script` arguments through the built-in `args` associative array. This feature can be leveraged to create parametric `Scripts` that dynamically adapt to the user's requests.

Whether a certain parameter can be specified as a literal, as an expression, or both, depends entirely on the instruction in which it is employed. This information, along with other useful details regarding the PerLa Scripting language, is available in the following instruction compendium.

4.3.1.1 `var` instruction

Description

Declares a new variable. This instruction requires two mandatory parameters:

- **name:** The variable name, namely a textual identifier used to reference the variable in later instructions. It must be unique in the scope of a single script.
- **type:** The type of data that can be stored inside the newly created variable. It can be set to one of the six PerLa primitive data types, or to a user-defined message type.

Usage examples

Creates a new variable named `count` of primitive type `integer`.

```
1 <i:var name="count" type="integer"/>
```

Creates a new variable named `cmd` of complex type `node_command`, whose declaration is omitted for brevity reasons.

```
1 <i:var name="cmd" type="node_command"/>
```

4.3.1.2 `set` instruction

Description

Sets the contents of a variable to a new value. The optional `field` parameter can be used whenever the user needs to modify a specific field in a variable of complex type.

- **variable:** Name of the variable to be modified.
- **field (optional):** An optional parameter that can be used to select the specific field to set in a variable of complex type.

- **value:** The new value of the variable. This parameter may be either a literal value or an expression.

Usage examples

Sets the previously defined `count` variable to the literal value `5`.

```
<i:set variable="count" value="5"/>
```

Sets the field `operation` of the previously defined `cmd` variable to the literal value `sample`.

```
<i:set variable="cmd" field="operation" value="sample"/>
```

Converts a temperature reading from Celsius to Fahrenheit degrees, and stores it in the `temp_f` field of a hypothetical variable named `result`.

```
<i:set variable="result" field="temp_f"
    value="${result.temp_c * 9/5 + 32}"/>
```

Deep copy. The content of the source variable is accessed with the "`${original}`" expression.

```
<i:set variable="copy" value="${original}"/>
```

4.3.1.3 `append` instruction

Description

Appends a new element to the end of a list-qualified field.

- **variable:** Name of the variable to be modified.
- **field:** Name of the list-qualified field to which the new value is to be appended.

- **value:** The new value to be inserted. This parameter may either be a literal value or an expression.

Usage examples

Appends the literal value “5” to a list field.

```
<i:append variable="result" field="temp_list" value="5"/>
```

4.3.1.4 `submit` instruction

Description

Submits an `IORequest` on a `Channel`. This instruction supports the following parameters:

- **request:** Identifier of the `IORequest` to be submitted.
- **channel:** Identifier of the `Channel` on which the request has to be submitted
- **variable (optional):** Name of the variable used to store the result of the I/O operation. If present, the complementary **type** parameter must be set. It is important to note that the result variable is automatically declared by the `submit` instruction; therefore, the final user must not create it with an explicitly `var` instruction.
- **type (optional):** Type of the variable used to store the result of the I/O operation. Its presence is subordinated to the aforementioned **type** parameter.

Additional `IORequest` parameters may be specified by supplying an appropriate list of `param` XML tags, each of which must contain the name of the parameter being set, and a reference to a variable containing the desired value (see the usage example section below for further syntax information). Upon submission, this instruction will automatically handle every `Mapper` operation required to convert the parameter value into a `Payload` object suited to the I/O operation.

Usage examples

Basic usage, submits the "start_sampling" request to a SerialChannel. All information received during the I/O operation is discarded, since no result variable is specified for this instruction.

```
<i:submit request="start_sampling" channel="serial"/>
```

Submits the "get_data" request to a HTTPChannel. All bytes received from the remote server are stored in the result variable.

```
<i:submit request="get_data" channel="http"
  variable="result" type="json_result"/>
```

Submits the "send_command" request to a SerialChannel. The command variable is set as an IORequest parameter.

```
<i:submit request="get_data" channel="http">
  <i:param name="payload" variable="command"/>
</i:submit>
```

4.3.1.5 stop instruction

Description

Stops the Script. This instruction is usually employed in conjunction with the if control structure to implement advanced halt conditions based on information available only at runtime.

Usage example

Immediately stops the execution of the Script.

```
<i:stop/>
```

Guarded stop. Halts the execution of the Script only when a certain condition holds true.

```
<i:if condition="${temp_c > 25}">
  <i:then>
    <i:stop/>
  </i:then>
</i:if>
```

4.3.1.6 `error` instruction

Description

Aborts the `Script`, signalling an abnormal execution condition. This instruction must be supplied with a **message** parameter that indicates the cause of failure. Similarly to the `stop` instruction, `error` invocations are commonly guarded by an `if` control structure to implement advanced error management behaviours.

Usage examples

The following code excerpt throws an error when the humidity level falls outside the acceptable range. This example combines the `stop` and `error` instructions to demonstrate a typical PerLa `Script` error management pattern.

```
<i:if condition="${humidity >= 0 && humidity <= 100}">
  <i:then>
    <i:put expression="${humidity}" attribute="humidity"/>
    <i:emit/>
    <i:stop/>
  </i:then>
  <i:else>
    <i:error message="humidity out of range"/>
  </i:else>
</i:if>
```

4.3.1.7 `put` instruction

Description

Adds a field into the staging area, viz. a temporary storage location used for the incremental creation of new output records. This instruction requires two mandatory parameters:

- **attribute:** Name of the attribute corresponding to the value being added in the staging area. The purpose of this parameter is twofold: first, it defines the name through which the new data can be retrieved (record field names always correspond to device attribute names); second, it is used to confirm that the value being added in the staging area has the correct data type (record field types always correspond to device attribute types).
- **expression:** Value of the new record field.

This instruction is intended to be called multiple times in the lifetime of a single `Script` execution, as a single `put` operation can only be used to set one record field at a time. As soon as all the desired values are staged, the content of the entire staging area can be flushed into a new record by means of the `emit` instruction.

It is worth noting that the content of the staging area is not deleted once `emit` is invoked. Though this may seem counter-intuitive or even undesirable, such behaviour allows for a simpler and more efficient management of aggregated data. The ability to retain all field values set with previous invocations of the `put` instruction is key to the example of figure 4.8, where low frequency information — namely the *name-id* and the *city* records — is set only once, and only the high-frequency samples are continuously replaced with new calls to the `put` instruction. This optimization technique would not be possible if the staging area were not provided with the aforementioned memory-retaining mechanism.

Usage Examples

Refer to section 4.3.1.8 for combined usage examples of the instructions `put` and `emit`.

4.3.1.8 `emit` instruction

Description

Creates a new record using the field values stored in the staging area. All records created by the `emit` instruction are released to the user only if the `Script` terminates without errors.

Usage Examples

Creates a record containing two literal fields.

```
<i:put attribute="temperature" expression="25"/>
<i:put attribute="humidity" expression="85"/>
<i:emit/>
```

Maps a flat data structure into a record. Differently from the previous example, record values are dynamically read from the `sample` variable, hypothetically received from a remote sensor node.

```
<i:put attribute="temperature" expression="${sample.temperature}"/>
<i:put attribute="humidity" expression="${sample.humidity}"/>
<i:put attribute="timestamp" expression="${sample.timestamp}"/>
<i:emit/>
```

`Script` expressions can also be used to create new record values at runtime. In the following code snippet, a simple conversion formula is employed to derive the `temp_fahrenheit` record field from other information sent by the sensor node.

```
<i:put attribute="temp_centigrade" expression="${sample.temp_cent}"/>
<i:put attribute="temp_fahrenheit"
  expression="${sample.temp_centigrade * 9/5 + 32}"/>
<i:put attribute="humidity" expression="${sample.humidity}"/>
<i:put attribute="timestamp" expression="${sample.timestamp}"/>
<i:emit/>
```

Looping over multiple samples stored in a list. The following example creates a new record for each element contained in the `data.samples` field.

```
<i:foreach items-var="data" items-field="samples" variable="sample">
  <i:put expression="${sample.timestamp}" attribute="timestamp"/>
  <i:put expression="${sample.temp}" attribute="temperature"/>
  <i:put expression="${sample.hum}" attribute="humidity"/>
  <i:emit/>
</i:foreach>
```

Exploiting the characteristic memory-retention feature of the `put` instruction to efficiently combine high-frequency and low-frequency information. Note that the `node` and `city` fields are staged only once, while all other fast changing information requires the execution of a `put` instruction for each list element.

```
<i:put expression="${input.node-id}" attribute="node"/>
<i:put expression="${input.city}" attribute="city"/>
<i:foreach items-var="input" items-field="samples" variable="sample">
  <i:put expression="${sample.timestamp}" attribute="timestamp"/>
  <i:put expression="${sample.temp}" attribute="temperature"/>
  <i:put expression="${sample.hum}" attribute="humidity"/>
<i:emit />
</i:foreach>
```

4.3.1.9 `if` control structure

Description

A conditional control structure for executing different `Script` branches depending on whether a user-specified **condition** expression evaluates to true or false.

Usage example

`If..then` example. Sets the variable `alarm` to TRUE when the temperature rises

above 25° C.

```
<i:if condition="${temp_c > 25}">
  <i:then>
    <i:set variable="alarm" value="true"/>
  </i:then>
</i:if>
```

`If..then..else` example. Sets the variable `tropical` to TRUE when the temperature rises above 30° C and the humidity is greater than 90%, to false otherwise.

```
<i:if condition="${temp_c > 25 && hum > 90}">
  <i:then>
    <i:set variable="tropical" value="true"/>
  </i:then>
  <i:else>
    <i:set variable="tropical" value="true"/>
  </i:else>
</i:if>
```

4.3.1.10 `foreach` control structure

Description

A control structure for traversing list-qualified `Message` fields. It can be used to repeat a given block of code for every element of a collection. The `foreach` control structure supports the following parameters:

- **items-var:** Name of the source variable.
- **items-field:** Name of the source field, namely the list-qualified field inside the **items-var** on which to loop over.
- **variable:** Name of the variable through which the current item is exposed.
- **index (optional):** Index of the current item.

Usage examples

Computing the average of all temperatures received from a remote sensor node.

```
<i:var name="count" type="integer"/>
<i:set variable="count" value="0"/>
<i:var name="avg" type="float"/>
<i:set variable="avg" value="0"/>
<i:foreach items-var="data" items-field="samples" variable="sample">
  <i:set variable="avg" value="${avg + sample.temperature}"/>
  <i:set variable="count" value="${count + 1}"/>
</i:foreach>
<i:set variable="avg" value="${avg / cont}"/>
```

4.3.2 Script Engine architecture and execution model

The `Script Engine` is the Middleware component responsible for the execution of PerLa `Scripts`. It is currently implemented as a program interpreter that parses and executes an intermediate PerLa `Script` representation generated by the `FPCFactory`, dubbed SIR (Script Intermediate Representation). SIR programs are directed graphs, where each node is an instruction, and each arc is a potential evolution of the program status; thus, SIR-encoded `Scripts` can be run by simply traversing the source data structure until a `stop` instruction is encountered, or an error is thrown. Thanks to this intermediate representation, the `Script Engine` architecture is lean and efficient; the core execution loop need not be concerned with the continuous interpretation of textual instructions or with complex error-checking procedures, as these two operations are only performed once, by the `FPCFactory`, when a `Script` is translated in its corresponding SIR form. Moreover, as a result of the additional decoupling provided by this intermediate code representation, the introduction of a new PerLa `Script` format does not entail any modification to the `Script Engine`, as long as a suitable SIR translator is provided for the new syntax.

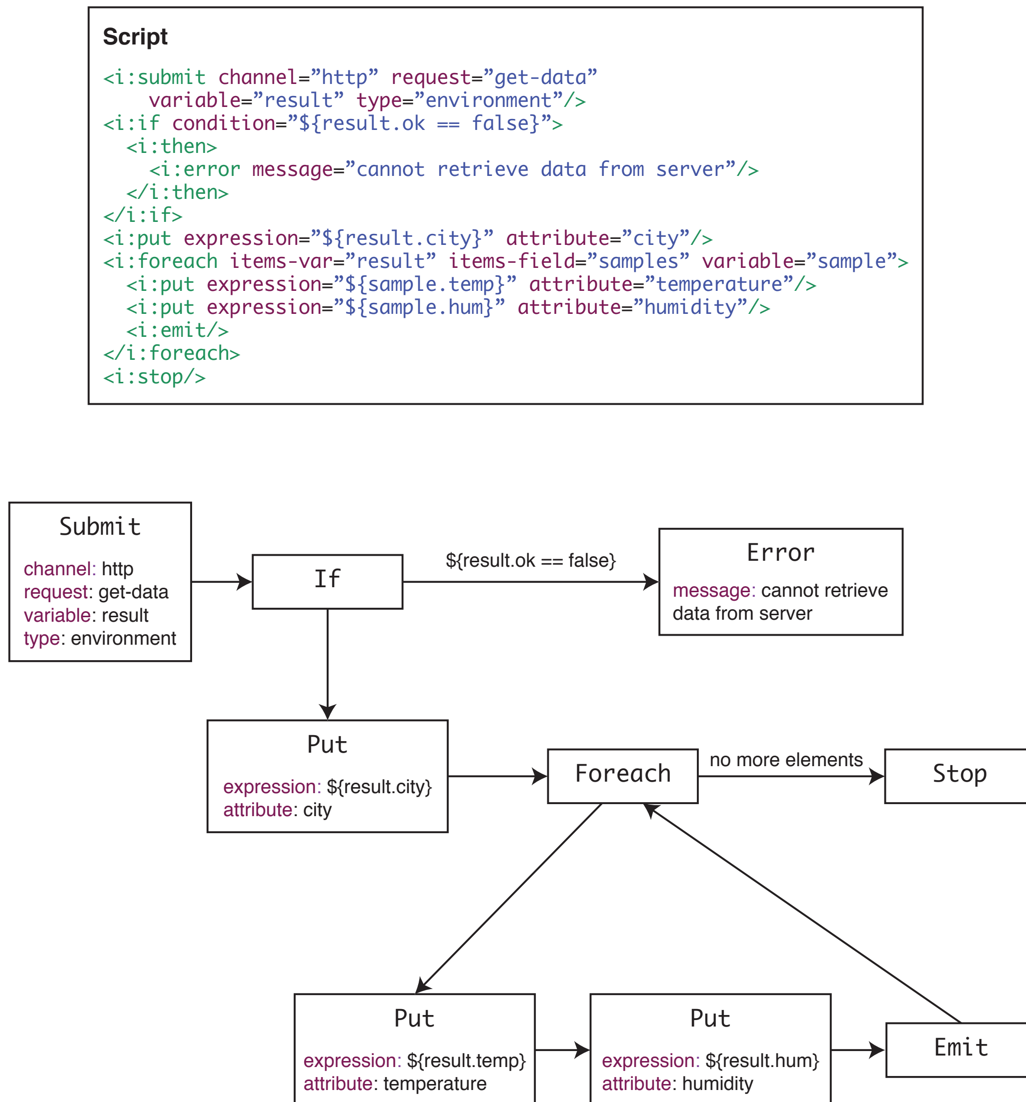

Figure 4.9: A PerLa `Script` and its corresponding SIR representation

Once started, PerLa `Scripts` are sandboxed in a dedicated thread of execution. For better isolation, the current status of each running `Script` instance is stored inside a private `ExecutionContext` object, which contains the following elements:

- **Program Counter:** A reference to the current instruction;
- **Variable Map:** An associative array that stores the current value of all

declared variables;

- **Record staging area:** Temporary working area used to create new records;
- **Output records:** List of records to be returned when a `stop` instruction is encountered.

This design ensures that all changes a single `Script` makes to its environment will remain private, that its status won't be altered by any other rogue routine, and that several `Scripts` can run simultaneously without interfering with each other.

The execution of a `Script` is always subordinated to an external event, like the submission of a new user request or the arrival of information from a sensing device. In particular, PerLa `Scripts` associated with the management of sensor data tend to run frequently and for a relatively short period of time, as their execution is triggered by each sample collected from the sensing network. To better cope with such usage scenario, the `Script Engine` implements a thread caching mechanism that reduces memory usage and startup times by reclaiming the runtime environment of each terminated `Script`, and re-purposing it for a new execution. This caching technique greatly reduces the overall number of objects allocated by the Java Virtual Machine, and guarantees that the overhead due to the initialization of new `ExecutionContext` instances is shared between multiple `Script` runs.

Moreover, the `Script Engine` can preemptively pause I/O bound computations to optimize the usage of available system resources. This feature, implemented by leveraging the asynchronous I/O design of the PerLa Middleware, is totally transparent to the `Script` developer, who should not worry with matters of concurrent programming; `Scripts` are automatically paused after an `IORequest` is submitted to a `Channel`, and their execution resumes as soon as the I/O activity terminates. These two operations — pause and resume — are performed within the `submit` instruction, which interrupts the `Script` after an I/O request is submitted, and restarts it once the associated response is available.

4.4 Putting it all together: the FPC

The FPC (Functionality Proxy Component) is the main data access interface available to the PerLa Middleware. Its chief duty is to provide a high-level abstraction of a Pervasive System by exposing the functionalities of all devices of the network through a single consistent API. Every instance of the PerLa Middleware hosts multiple FPCs, one for each sensing node. This one-to-one relationship — the fulcrum of the PerLa philosophy — is a fundamental architectural feature that endows the Middleware with utmost flexibility and the finest granularity of control, as it presents final users with the possibility to manage heterogeneous networks of sensing devices, down to the single node, through a uniform set of high-level functions.

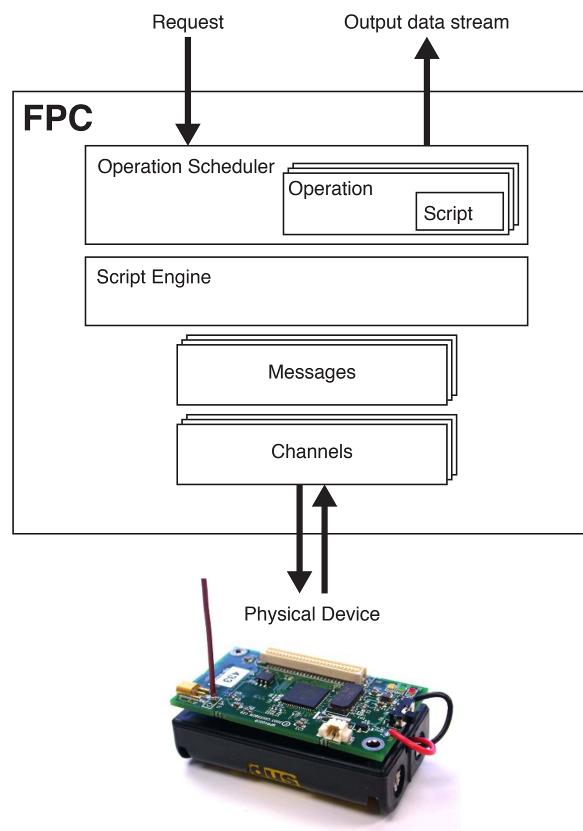

Figure 4.10: Internal structure of the Functionality Proxy Component

4.4.1 Accessing device features

FPCs are created from the composition of all Middleware components described in the former sections of this chapter. As shown in figure 4.10, these software modules are complemented by a set of Operations and an Operation Scheduler. An Operation is a collection of Scripts committed to the management of a well defined aspect of an endpoint device, such as the retrieval of a specific data sample at periodic intervals of time. In total, there are four different Operation types available in the PerLa Middleware, each of which corresponds to a FPC action (get, set, periodic sampling and asynchronous event handling). Each of these Operations is associated with the list of device Attribute that can be modified or generated with it; this list is automatically inferred by the PerLa Middleware by analyzing the associated data access Scripts.

4.4.1.1 Get Operation

A single Script that retrieves information from the remote device. It can be used to perform a single-shot sampling operation or to read software parameters stored on the connected endpoint.

This Operation type is introduced by the <get> XML tag, and contains a single PerLa Script that is responsible for connecting with the remote device, retrieving the information requested by the user, and creating an output record. The following example shows a textbook implementation of the Get Operation, which demonstrates how all the aforementioned operations can be performed in a few lines of PerLa Script.

```
1 <get id="single-temp-sample">
2   <i:submit request="temperature-request" channel="serial"
3             variable="result" type="temperature-msg"/>
4   <i:put expression="${result.temperature}" attribute="temperature"/>
5   <i:emit/>
6 </get>
```

4.4.1.2 Set Operation

A single `Script` that sends information to the controlled device. It can be used to dispatch commands, activate mechanisms, or change software parameters on a remote sensing node.

All `Set Operation Scripts` must be enclosed in a `<set>` XML tag, and are required to define a series of actions resulting in the transmission of data to the remote device. An example of this `Operation` is available in the code excerpt below, which retrieves the current time from a parameter passed by the `FPC` user (`arg['timestamp']`), and delivers it by means of a `SerialChannel`.

```
<set id="set-clock">
  <i:create variable="settings" type="settings-msg"/>
  <i:set variable="settings" field="time" value="${arg['timestamp']}"/>
  <i:submit request="send-settings" channel="serial">
    <i:param name="payload" variable="settings"/>
  </i:submit>
</set>
```

4.4.1.3 Periodic Operation

A collection of `Scripts` for managing an unattended, periodic stream of information. The `Periodic Operation` is more complicated than previous `Operation` types, both from a syntactic and an operative point of view, as it requires the device developer to specify three different classes of `Scripts`.

The first of these is the `<start> Script`, which is employed to initialize the sampling operation. It normally contains a series of instruction that parse the signal rate requested by the user, configure the device to start the sampling operation, and handle potential error conditions. By default, the sampling period exposed to the `Script` by means of the `arg['period']` argument is expressed in milliseconds; it is the developer's duty to convert this value into whichever format is required

by the remote device. The code extract available below is worthy of note, since it is employed to initialize two different sampling operations at once, one for temperature, and one for humidity.

After the sampling operation is correctly initialized, the sensing node will begin sending data packets towards its controlling `FPC`. All the instructions required to convert these raw information into a record suitable for further processing have to be specified inside an `<on>` tag. As shown below, each `Periodic Operation` is to be equipped with an `<on> Script` for each different type of `Message` sent by the device.

Finally, the `Periodic Operation` is required to contain a `<stop> Script` that can be used to terminate the sampling operation and undo any action performed at startup.

```
<periodic id="weather-periodic">
  <start>
    <i:create variable="period" type="sampling-period"/>
    <i:set variable="period" field="period" value="${arg['period']}"/>
    <i:submit request="temperature-request" channel="simulator">
            <i:param name="period" variable="period"/>
    </i:submit>
    <i:submit request="humidity-request" channel="simulator">
            <i:param name="period" variable="period"/>
    </i:submit>
  </start>
  <stop>
    <i:create variable="period" type="sampling-period"/>
    <i:set variable="period" field="period" value="0"/>
    <i:submit request="temperature-request" channel="simulator">
            <i:param name="period" variable="period"/>
    </i:submit>
    <i:submit request="humidity-request" channel="simulator">
            <i:param name="period" variable="period"/>
```

```
20      </i:submit>
21    </stop>
22    <on message="temperature-msg" variable="result">
23      <i:put expression="${result.temperature}" attribute="temperature" />
24      <i:emit />
25    </on>
26    <on message="humidity-msg" variable="result">
27      <i:put expression="${result.humidity}" attribute="humidity" />
28      <i:emit />
29    </on>
30  </periodic>
31  </periodic>
```

4.4.1.4 Async Operation

A collection of `Scripts` that handles an asynchronous stream of information from the device, i.e. a series of events that are received at irregular intervals of time. Similarly to the `Periodic Operation`, the `Async Operation` features a `<start>` `Script` (optional), and a `<on>` `Script` for each different type of event message that may be received from the sensing node.

```
1   <async id="event-async">
2     <start>
3       <i:create variable="period" type="sampling-period"/>
4       <i:set variable="period" field="period" value="200"/>
5       <i:submit request="event-request" channel="simulator">
6         <i:param name="period" variable="period"/>
7       </i:submit>
8     </start>
9     <on message="event-msg" variable="result">
10      <i:put expression="${result.event}" attribute="event"/>
11      <i:emit />
```

```
12   </on>
13 </async>
```

4.4.2 The FPC interface

The `FPC` interface, whose signature is available in listing 4.13, represents one of the defining features of the PerLa Middleware. Its technology-agnostic data access methods provide an easy and intuitive way to access the information generated by a Pervasive System, and require no knowledge of the underlying hardware layer in order to be used.

```
1  public interface Fpc {
2      public int getId();
3
4      public String getType();
5
6      public Collection<Attribute> getAttributes();
7
8      public Task set(Map<Attribute, Object> valueMap, TaskHandler
           handler);
9
10     public Task get(Collection<Attribute> attributes, TaskHandler
           handler);
11
12     public Task periodic(Collection<Attribute> attributes, long periodMs,
13             TaskHandler handler);
14
15     public Task async(Collection<Attribute> attributes, TaskHandler
           handler);
16 }
```

Listing 4.13: The FPC interface

The first three methods of this interface — `getId()`, `getType()` and `getAttributes()` — are designed to retrieve a series of basic information concerning the remote endpoint. The first one, `getId()`, returns a numeric identifier that can be used to address the single specific node connected to the current `FPC` object. The second one, `getType()`, is employed to retrieve a brief textual description of the remote endpoint. Lastly, the `getAttributes()` method returns a comprehensive list of all device `Attributes` that can be sampled or modified using an `FPC`, qualified in terms of name, data type (`id`, `integer`, `float`, `string`, `boolean` or `timestamp`) and access permissions (`read-only`, `read-write` or `write-only`).

`Attribute` values can be retrieved or set using the `FPC`'s data access methods, namely `get()`, `periodic()`, `set()`, and `async()`, each of which correspond to a specific type of `Operation`. Similarly to what already seen for other Middleware components described in this document, all these methods implement the asynchronous interaction paradigm introduced in section 3.4. As shown in listing 4.13, their immediate return type is in fact a `Task` object, which can be employed to control the status of the ongoing data access operation. The actual data samples and events generated by the `FPC` are notified asynchronously through a `TaskHandler` using the following methods:

- **complete():** Signals that the operation associated with the `TaskHandler` has just been completed. It is employed to notify the successful completion of a `set()` operation, or to indicate that a `get()` operation has been stopped and will not produce any new record;
- **newRecord():** Delivers a new record;
- **error():** Indicates that the operation associated to the `TaskHandler` was aborted due to an error.

```
public interface Task {
    public Collection<? extends Attribute> getAttributes();

    public boolean isRunning();

    public void stop();
}

public interface TaskHandler {
    public void complete(Task task);

    public void newRecord(Task task, Record record);

    public void error(Task task, Throwable cause);
}
```

Listing 4.14: The `Task` and `TaskHandler` interfaces.

The user is always required to list all `Attributes` to be sampled when invoking one of the available data retrieval methods; this information is employed by the `FPC` to check whether the requested information can be gathered through the remote device or not, and to select an `Operation` that best suits the user's demands. Both these activities are performed by the `OperationScheduler`, a Middleware component tasked with managing all data handling `Operations` available in an `FPC`. This component is also responsible for the management of concurrent operations occurring at different sampling rates, a common use case that requires the `OperationScheduler` to start a single periodic `Operation` using the highest requested sampling frequency, and to distribute the resulting records according to the requirements of each single user.

An additional thing of note regarding the `OperationScheduler` is its ability perform several types of `Operations` even if the endpoint device does not support them natively; periodic sampling activities can be simulated from a simple `Get`

Operation simply by executing a single-shot request at regular intervals, whereas starting a Periodic Operation and stopping it after the arrival of the first data sample is tantamount to a Get Operation.

4.4.3 FPC Factory

As suggested by its name, the FPCFactory is the Middleware component responsible for creating new FPC objects. It plays a fundamental role in the Middleware's Plug-and-Play device addition mechanism, since its primary task consists in the creation and final assembly of all constituent parts of an FPC object.

The creation of an FPC is a complex process that begins with the reception of a Device Descriptor, a declarative document containing a machine-parseable blueprint of the newly connected sensor node. Device Descriptors are composed of several parts, the contents of which have been thoroughly explained in previous sections of this document:

- **Device Attributes:** Declaration of all data items supported by the device;
- **Channels and Requests:** Configuration of all Channels and IORequest objects required to communicate with the endpoint device;
- **Messages:** Contains the declaration of all data structures employed during the communication with the endpoint device, along with the strategies to be followed for serializing and deserializing high-level information into Channel-ready Payload objects;
- **Operations:** Definition of the PerLa Script procedures that are to be used for interacting with the remote device.

During the installation of a new PerLa instance, users must guarantee that every available Channel has the possibility to relay new Device Descriptors towards an active FPCFactory; failure to do so may prevent some portions of the sensing network from establishing an autonomous connection with the Middleware. This configuration is to be done according to the characteristics of each specific Channel, since every communication mechanism may enforce a different connection

paradigm. For example, the `HTTPChannel` module can't be connected to the `FPCFactory`, as web services and REST APIs are not able to initiate a spontaneous connection to the Middleware, and thus can't send their Device Descriptors; `SerialChannels`, on the other hand, expose a dedicated handler method designed to broadcast Device Descriptor data to any interested `FPCFactory`.

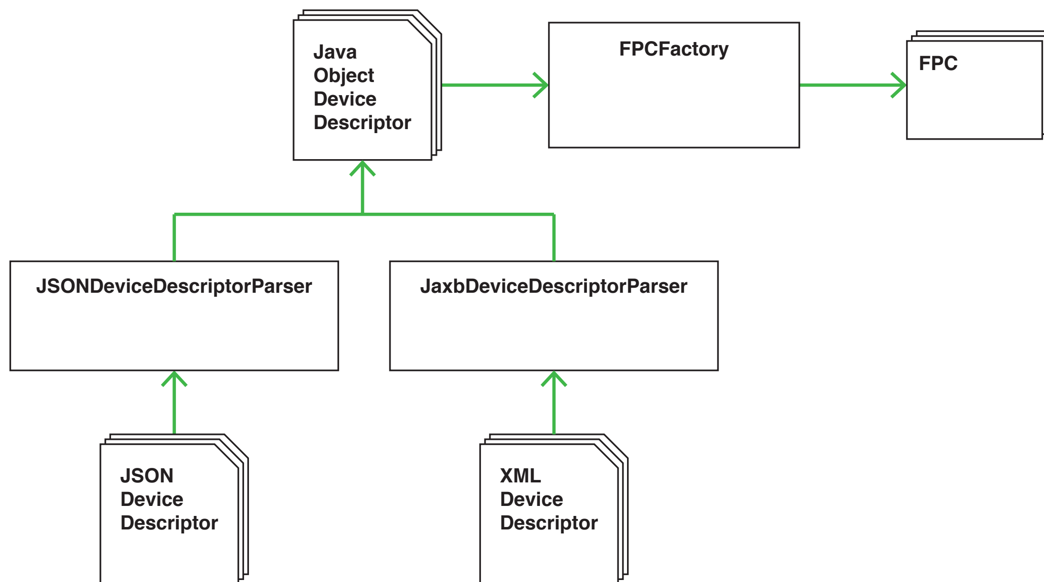

Figure 4.11: Translation of different Device Descriptor formats into an intermediary Java representation

Figure 4.11 shows a section of a hypothetical Middleware setup. As can be seen, Device Descriptors are parsed and transliterated into an intermediate Java representation before being handed over to the `FPCFactory`. This decoupling process, performed by the `DeviceDescriptorParser` interface, ensures that the `FPCFactory` is not directly tied to any particular Device Descriptor format; as a result, the introduction of a new descriptor syntax does not entail any change to the `FPCFactory` itself, but only the creation of an appropriate `DeviceDescriptorParser` class. At the present moment, the PerLa Middleware includes a `JaxbDeviceDescriptParser` implementation, which is responsible for translating the XML Device Descriptor syntax shown in previous sections of this document into the corresponding `DeviceDescriptor` Java class required by the `FPCFactory`.

```java
public interface DeviceDescriptorParser {
    public DeviceDescriptor parse(InputStream is)
            throws DeviceDescriptorParseException;
}
```

Listing 4.15: The DeviceDescriptorParser interface. The only method exposed by this module is responsible for converting Device Descriptors received from the sensing nodes into an intermediate Java object representation.

The `FPCFactory`, in its essence, is a coordinator object that delegates the actual construction of all `FPC` components to the various factory modules described in the former sections of this chapter. This feature is an essential characteristics of the new Middleware architecture, and constitutes the basic building block of the PerLa pluggable module system. New `Channels`, `IORequests`, and `Messages` can be added simply by means of the `FPCFactory` constructor method (see listing 4.16), and don't require any modification to the existing Middleware code; PerLa is open for extension but closed for modification.

In addition to the creation of all `FPC` component modules, the `FPCFactory` performs the following tasks:

- Parses the `<operation>` Device Descriptor section and compiles all PerLa `Scripts` in their intermediate SIR form;
- Associates every declared device `Attribute` to the `Operation` that is responsible for its management;
- Assembles the various component modules, `Operations` and `Scripts` inside a single `FPC` object.

```java
public class FPCFactory {
   public BaseFpcFactory(List<MapperFactory> mapperFactoryList,
```

```
3             List<ChannelFactory> channelFactoryList,
4             List<IORequestBuilderFactory> requestBuilderFactoryList)
5
6      public Fpc createFpc(DeviceDescriptor descriptor, int id)
7             throws InvalidDeviceDescriptorException;
8  }
```

Listing 4.16: The FPCFactory class methods

4.4.4 Registry

The `Registry` is a simple in-memory database that stores `FPC` objects. It is primarily employed for the discovery of sensing devices registered in a running PerLa instance, and its services are extensively exploited by the Query Executor component for the management of `EXECUTE IF` statements. Its interface, available in listing 4.17, is straightforward and easy to use. It is composed of two data manipulation methods, namely `add()` and `remove()`, and three data retrieval methods, `get()`, `getAll()` and `getByAttribute`.

`add()` and `remove()` allow PerLa users to respectively insert and delete `FPC` objects from the `Registry`. The first method is primarily used by the `FPCFactory`, which is responsible for registering newly connected devices, whereas the second is invoked from the `FPC` itself when the controlled node stops operating.

The data retrieval section of the `Registry` interface comprises three methods with different semantics. `get()` can be used to retrieve a single `FPC` object with a specific identifier, and is usually invoked when the user needs to address a certain device in a Pervasive Network. `getAll()` is a shortcut that returns a list with all the `FPC` objects connected to PerLa. The last method, `getByAttribute()`, allows PerLa users to select a subset of devices with well-defined characteristics; its two parameters can in fact be used to indicate the precise set of data `Attributes` that a remote node must possess in order to be considered in the selection.

```java
public interface Registry {
    public Fpc get(int id);

    public Collection<Fpc> getAll();

    public Collection<Fpc> getByAttribute(Collection<Attribute> with,
        Collection<Attribute> without);

    public void add(Fpc fpc);

    public void remove(Fpc fpc);
}
```

Listing 4.17: The Registry interface

Chapter 5

Conclusions

This thesis described the design and implementation of an asynchronous data access middleware for Pervasive Systems. As shown in previous chapters, this process began with an analysis aimed at identifying the weaknesses and strengths of the Classic PerLa Middleware architecture, which was later used to outline a basic set of goals to be followed during the development of the software hereby described. From these goals ensued a New Middleware design (chapter 3), and a concrete implementation (chapter 4). The most important contributions that the development of this new data access middleware brought to the PerLa System can be classified into three categories: modularization of the Plug & Play device registration process, asynchronous data flow management, and an improved `FPC`.

The new Plug & Play device registration process was enhanced with three distinct measures: first, the internal structure of the `FPC` component was split into independent modules; second, the `FPCFactory` itself was partitioned into several sub-factory units, one for each `FPC` module; third, a Plugin System was designed to allow the addition of new `FPC` fragments without requiring any direct modification to the `FPCFactory` itself. The advantages and merits of this new modular design were tested and validated with the implementation of five different modules, three created by the author of this thesis (`JSONMapper`, `URLEncodedMapper` and `SimulatorChannel`), and two by other graduate student (`HTTPChannel` and

TinyOSChannel).

Another crucial aspect of the New PerLa Middleware design is represented by the asynchronous data flow management paradigm. As explained in section 3.4, all components of the new architecture implement an asynchronous event-driven API that improves both memory usage and global reaction times of the entire system. An example of the benefits brought by this new paradigm can be derived by analysing the number of threads instantiated for each FPC. In the Classic Middleware, each FPC was composed of four distinct Java threads: one dedicated to reading incoming messages, one for the Unmarshaller, one for the Marshaller, and one for the creation of output records. Conversely, within the New Middleware infrastructure, a single Java thread located in the Channel module is enough to drive all data handoffs occurring inside an entire FPC. Although additional threads may be instantiated during the execution of PerLa Scripts, it is worth noting that these are managed by the Script Engine, and are always shared among all running FPCs; as a consequence, the New Middleware can dynamically adjust its resource consumption figures to match the actual workload (see chapter 4 for additional details).

The FPC benefits from another improvement brought by the New Middleware design, namely the PerLa Scripting Language. This new feature complements the declarative nature of the Device Descriptor, enabling the definition of advanced mappings between device capabilities and data Attributes exposed by the FPC component. The imperative programming paradigm fostered by the PerLa Scripting Language proved to be a key addition to the Middleware architecture, as it enhanced the flexibility and versatility of the entire PerLa System; Scripts have in fact been used to define complex device initialization procedures, to aggregate the information collected from a sensing network, and to reshape the contents of hierarchical data structures into the one-dimensional record pattern produced by the FPC component. It is in the author's view that the former Device Descriptor structure could not be used to adequately model the aforementioned applications, as its inherently declarative essence embodied a fixed set of behavioural assump-

tion which were forced on all nodes of the Pervasive Network. The PerLa Scripting Language proposes itself as a less opinionated tool that can be used by node developers to better specify the functioning mechanisms of their devices.

5.1 Future work

5.1.1 Implementation of new plugins

The New PerLa Middleware is designed to be extended through the addition of new modules, and it should come as no surprise that one of its intended evolution paths consists in fact in the development of new Plugins. As described in chapter 4, there are two main types of modules that can be added to the PerLa Plugin System: `Channels` and `Mappers`.

The possibility to add new `Channel` implementations is a distinguishing feature of the New Middleware design that should be effectively exploited to expand the range of supported endpoint devices. At the time of writing the selection of Plugins shipped with the core Middleware distribution allows PerLa to connect with HTTP services and TinyOS motes. This initial line-up should be only considered as a starting point, since a larger assortment of communication systems is required to manage even the most rudimentary Pervasive System. The following list contains a choice of protocols and networking technologies for which a dedicated `Channel` implementation is advised:

- **TCP/IP:** Widely employed in a vast variety of devices, ranging from high-end personal computers to low power devices;
- **Bluetooth LE (Low Energy):** One of the leading technologies for wireless personal area networks. Currently supported by all major operating systems, Bluetooth LE found its way into many devices and appliances, like fitness bands, smartphones, healthcare instruments, home entertainment sets and localization beacons;

- **IEEE 802.15.4 based protocols:** IEEE 802.15.4 is a physical layer widely employed in many personal area network protocols. It is the foundation of several networking specifications like Zigbee, Xbee and MiWi;
- **RS232:** Serial port communication. Its implementation should be considered in order to connect with legacy devices and other low power systems (e.g., Arduino).

`Mapper`s represent another extension point of the PerLa Middleware infrastructure that can be used to manage additional data formats and encodings. The only `Mapper` components available as of December 2014 in the core Middleware distribution provide support for JSON and URL-encoded data structures. Analogously to what already stated for the `Channel` component, future PerLa developers should consider implementing new `Mapper`s for handling the following data formats:

- **C/C++ structs:** Its implementation should be a simple backport from the Classic Middleware architecture;
- **XML:** A markup language for document encoding;
- **CSV (Comma-Separated Values):** A simple format for the transmission and storage of tabular data.

5.1.2 Alternative Device Descriptor forms

As introduced in chapter 4, the new Plug & Play node registration system is comprised of two separate elements: a `DeviceDescriptorParser` front-end, which analyzes the Device Descriptor to build a format-agnostic Java representation of the descriptor itself, and the `FPCFactory`, which consumes this intermediate Java representation to assemble the final `FPC`. This new architecture was conceived to facilitate the future addition of alternative Device Descriptor formats by only requiring the development of an appropriate `DeviceDescriptorParser` module.

There are two main reasons for adding a new Device Descriptor representation: first, to support a different data format that may be more convenient for some devices (e.g., a JSON Device Descriptor); second, to create `FPC`s using readily

available industry standard device description technologies. This former motivation leads us to one future development of the PerLa infrastructure, namely the possibility of introducing a `DeviceDescriptorParser` for the *SensorML* [1] sensor description format. Through this effort the PerLa Middleware would be immediately compatible with all devices for which a SensorML description is already available.

5.1.3 Distributed PerLa

Future development of the PerLa Middleware should aim at implementing in-network processing capabilities in order to better exploit the resources available in a Pervasive System. Such efforts must focus on the definition of a software distribution infrastructure that can be used to divide a high-level computation into smaller, independent units of work to be executed on the individual nodes of the sensing network.

Bibliography

[1] Sensor model language (sensorml). *OpenGIS Implementation Specification OGC*, 2013. http://www.opengeospatial.org/standards/sensorml.

[2] Karl Aberer, Manfred Hauswirth, and Ali Salehi. The global sensor networks middleware for efficient and flexible deployment and interconnection of sensor networks. Technical report, 2006. http://infoscience.epfl.ch/record/83891, sumitted to ACM/IFIP/USENIX 7th International Middleware Conference.

[3] 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.

[4] David Chu, Arsalan Tavakoli, Lucian Popa, and Joseph Hellerstein. Entirely declarative sensor network systems. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1203–1206. VLDB Endowment, 2006.

[5] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[6] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[7] Marco Fortunato and Marco Marelli. Design of a declarative language for pervasive systems, 2006/2007. MSc Thesis.

[8] Siemens Gmbh. Sword, 2006. http://webdoc.siemens.it/CP/SIS/Press/SWORD.htm, internal communication.

[9] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. Firewxnet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 28–41. ACM, 2006.

[10] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *ACM Sigplan Notices*, volume 37, pages 96–107. ACM, 2002.

[11] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[12] Thomas Luckenbach, Peter Gober, Stefan Arbanowski, Andreas Kotsopoulos, and Kyle Kim. Tinyrest: A protocol for integrating sensor networks into the internet. In *Proc. of REALWSN*, pages 101–105. Citeseer, 2005.

[13] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[14] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM, 2002.

[15] 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, 2012.

[16] 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, 2012.

[17] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

[18] Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 108–120. IEEE, 2005.

Appendix A

Complete XML Device Descriptor examples

A.1 Example 1

The Device Descriptor portrayed in this first example is employed to create a Simulator FPC, a useful tool that helps testing the PerLa Middleware even when no physical sensor nodes are available. This simulated device exposes two attributes: a read-only float representing the temperature in Celsius degrees (temp_c), and a write-only integer that will be used to set the sampling period (period).

This FPC contains one Channel of type SimulatorChannel, which generates new data samples without requiring a connection to a real sensing device. In our example, the SimulatorChannel is configured with a single data generation routine, named temperature, that automatically creates new data samples in increments of 0.1°, spanning from a minimum of 16° C to a maximum of 20° C. The <request> section shows that a single IORequest object is enough to drive the single data generator available in this FPC.

Only two Message types are declared in the Device Descriptor: a temperature-msg message, which contains a single field of type float, and a sampling-period

message, composed of only an `integer` field. These two messages will be used to collect temperature samples and to set the desired sampling period in the `SimulatorChannel` respectively.

As can be seen from the `<operation>` section of this descriptor, the simulator device exposes a periodic sampling operation named `temp-periodic`. It is important to note that the `start Script` initializes the `SimulatorChannel` according to the sampling rate specified by the user, which is retrieved with a `${arg['period']}` expression. New output records are create by the `on Script` upon arrival of each `temperature-msg`. Finally, the generation of new data samples can be stopped by setting the sampling frequency to zero.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <device type="Weather simulator"
3      xmlns="http://perla.dei.org/device"
4      xmlns:i="http://perla.dei.org/device/instructions"
5      xmlns:sim="http://perla.dei.org/channel/simulator">
6
7    <attributes>
8      <attribute id="temp_c" type="float" permission="read-only"/>
9      <attribute id="period" type="integer" permission="write-only"/>
10   </attributes>
11
12   <channels>
13     <sim:channel id="simulator">
14       <sim:generator id="temperature">
15         <sim:field name="temperature" strategy="step"
16                   type="float" min="16" max="20" increment="0.1"/>
17         </sim:generator>
18     </sim:channel>
19   </channels>
```

```
20
21  <messages>
22    <sim:message id="temperature-msg">
23      <sim:field name="temperature" type="float"/>
24    </sim:message>
25    <sim:message id="sampling-period">
26      <sim:field name="period" type="integer"/>
27    </sim:message>
28  </messages>
29
30  <requests>
31    <sim:request id="temperature-request" generator="temperature"/>
32  </requests>
33
34  <operations>
35    <periodic id="temp-periodic">
36      <start>
37        <i:create variable="period" type="sampling-period"/>
38        <i:set variable="period" field="period"
              value="${arg['period']}"/>
39        <i:submit request="temperature-request" channel="simulator">
40          <i:param name="period" variable="period"/>
41        </i:submit>
42      </start>
43      <stop>
44        <i:create variable="period" type="sampling-period"/>
45        <i:set variable="period" field="period" value="0"/>
46        <i:submit request="temperature-request" channel="simulator">
47          <i:param name="period" variable="period"/>
48        </i:submit>
49      </stop>
50      <on message="temperature-msg" variable="result">
51        <i:put expression="\${result.temperature}" attribute="temp_c" />
```

```
52        <i:emit />
53      </on>
54    </periodic>
55  </operations>
56
57 </device>
```

A.2 Example 2

This second Device Descriptor is used to configure an `FPC` for collecting data from a RESTful meteorological service. The remote API is accessed by means of a `HTTPChannel`, which retrieves a complete JSON-encoded weather report for the city of Milan.

The service chosen for this application returns its results using a fairly complex JSON entity, whose structure has been replicated in the `<message>` section of this descriptor. The main `weather` object is in fact a wrapper for three other components: a `coord` message, a `data` message, and a `wind` message. It is worth noting that these three sub-objects are constructed using only primitive fields, and that they are not intended to be used on their own; their only purpose is to be included as the constituent elements of a `weather` message.

The stateless nature of the HTTP protocol is underscored by the simplicity of the `HTTPChannel` declaration, which doesn't require any parameter except for its own identifier, and by the related `HTTPIORequest`, where all information required for querying the RESTful API endpoint is stored. These two `FPC` components are then employed in the `<get> Operation`, which is tasked with creating a new record from the `weather` object retrieved from the web service.

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <device type="REST Weather station"
3      xmlns="http://perla.dei.org/device"
4      xmlns:js="http://perla.dei.org/fpc/message/json"
5      xmlns:http="http://perla.dei.org/channel/http"
6      xmlns:i="http://perla.dei.org/device/instructions">
7
8    <attributes>
9      <attribute id="city" type="string" permission="read-only"/>
10     <attribute id="temp_k" type="float" permission="read-only"/>
11     <attribute id="temp_c" type="float" permission="read-only"/>
12     <attribute id="temp_f" type="float" permission="read-only"/>
13     <attribute id="pressure" type="float" permission="read-only"/>
14     <attribute id="humidity" type="float" permission="read-only"/>
15     <attribute id="wind_speed" type="float" permission="read-only"/>
16     <attribute id="wind_deg" type="float" permission="read-only"/>
17   </attributes>
18
19   <channels>
20     <http:channel id="http"/>
21   </channels>
22
23   <messages>
24     <js:object id="coord">
25       <js:value name="lon" type="string"/>
26       <js:value name="lat" type="string"/>
27     </js:object>
28
29     <js:object id="data">
30       <js:value name="temp" type="float"/>
31       <js:value name="pressure" type="float"/>
32       <js:value name="humidity" type="float"/>
33       <js:value name="temp_min" type="float"/>
```

```
34        <js:value name="temp_max" type="float"/>
35      </js:object>
36
37      <js:object id="wind">
38        <js:value name="speed" type="float"/>
39        <js:value name="deg" type="float"/>
40      </js:object>
41
42      <js:object id="weather">
43        <js:value name="coord" type="coord"/>
44        <js:value name="data" type="data"/>
45        <js:value name="wind" type="wind"/>
46      </js:object>
47    </messages>
48
49    <requests>
50      <http:request id="weather-mi"
51                    host="http://api.openweathermap.org/data/2.5/weather?q=Milan,it"
52                    method="get" />
53    </requests>
54
55    <operations>
56      <get id="weather-mi">
57        <i:submit request="weather-mi" channel="http"
58                  variable="result" type="weather"/>
59        <i:put expression="Milan" attribute="city"/>
60        <i:put expression="${result.data.temp}" attribute="temp_k"/>
61        <i:put expression="${result.data.temp - 272.15}"
              attribute="temp_c"/>
62        <i:put expression="${(result.data.temp - 273.15) * 9 / 5 + 32}"
63                attribute="temp_f"/>
64        <i:put expression="${result.data.pressure}" attribute="pressure"/>
65        <i:put expression="${result.data.humidity}" attribute="humidity"/>
```

```
66        <i:put expression="${result.wind.speed}" attribute="wind_speed"/>
67        <i:put expression="${result.wind.deg}" attribute="wind_deg"/>
68        <i:emit/>
69      </get>
70    </operations>
71
72  </device>
```

A.3 Example 3

A purely educational Device Descriptor designed to showcase the following PerLa Middleware features:

- read-write, read-only write-only and static attributes;
- `SimulatorChannel` value generator configurations for all available data types;
- `Get` and `Set Operations`;
- `Periodic Operations` with single and multiple `<on>` `Scripts`.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <device type="test"
3       xmlns="http://perla.dei.org/device"
4       xmlns:i="http://perla.dei.org/device/instructions"
5       xmlns:sim="http://perla.dei.org/channel/simulator">
6
7     <attributes>
8       <attribute id="integer" type="integer" permission="read-write"/>
9       <attribute id="float" type="float" permission="read-write"/>
10      <attribute id="boolean" type="boolean" permission="read-write"/>
11      <attribute id="string" type="string" permission="read-write"/>
12      <attribute id="event" type="boolean" permission="read-only"/>
```

```
13    <attribute id="period" type="integer" permission="write-only"/>
14    <attribute id="static" type="integer" access="static" value="5"/>
15  </attributes>
16
17  <channels>
18    <sim:channel id="simulator">
19      <sim:generator id="all">
20        <sim:field name="type" strategy="static" value="all"/>
21        <sim:field name="integer" strategy="dynamic"
22                  type="integer" min="12" max="32"/>
23        <sim:field name="float" strategy="dynamic"
24                  type="float" min="450" max="600"/>
25        <sim:field name="string" strategy="dynamic"
26                  type="string" min="10" max="15"/>
27      </sim:generator>
28      <sim:generator id="integer">
29        <sim:field name="type" strategy="static" value="integer"/>
30        <sim:field name="integer" strategy="dynamic"
31                  type="integer" min="47" max="58"/>
32      </sim:generator>
33      <sim:generator id="string">
34          <sim:field name="type" strategy="static" value="integer"/>
35          <sim:field name="string" strategy="dynamic"
36                    type="string" min="5" max="5"/>
37      </sim:generator>
38      <sim:generator id="boolean">
39        <sim:field name="type" strategy="static" value="boolean"/>
40        <sim:field name="boolean" strategy="dynamic" type="boolean"/>
41      </sim:generator>
42      <sim:generator id="event">
43        <sim:field name="type" strategy="static" value="event"/>
44        <sim:field name="event" strategy="dynamic" type="boolean"/>
45      </sim:generator>
```

```
46      </sim:channel>
47    </channels>
48
49    <messages>
50      <sim:message id="set-msg">
51        <sim:field name="test" type="integer"/>
52      </sim:message>
53      <sim:message id="sampling-period">
54        <sim:field name="period" type="integer"/>
55      </sim:message>
56      <sim:message id="all-msg">
57        <sim:field name="type" type="string" qualifier="static"
             value="all"/>
58        <sim:field name="integer" type="integer"/>
59        <sim:field name="float" type="float"/>
60        <sim:field name="string" type="string"/>
61      </sim:message>
62      <sim:message id="integer-msg">
63        <sim:field name="type" type="string" qualifier="static"
             value="integer"/>
64        <sim:field name="integer" type="integer"/>
65      </sim:message>
66      <sim:message id="string-msg">
67        <sim:field name="type" type="string" qualifier="static"
             value="string"/>
68        <sim:field name="string" type="string"/>
69      </sim:message>
70      <sim:message id="boolean-msg">
71        <sim:field name="type" type="string" qualifier="static"
             value="boolean"/>
72        <sim:field name="boolean" type="boolean"/>
73      </sim:message>
74      <sim:message id="event-msg">
```

```
75        <sim:field name="type" type="string" qualifier="static"
              value="event"/>
76        <sim:field name="event" type="boolean"/>
77      </sim:message>
78    </messages>
79
80    <requests>
81      <sim:request id="all-request" generator="all"/>
82      <sim:request id="integer-request" generator="integer"/>
83      <sim:request id="string-request" generator="string"/>
84      <sim:request id="boolean-request" generator="boolean"/>
85      <sim:request id="event-request" generator="event"/>
86    </requests>
87
88    <operations>
89      <get id="integer-get">
90        <i:submit request="integer-request"
91                  channel="simulator" variable="result" type="integer-msg"/>
92        <i:put expression="${result.integer}" attribute="integer"/>
93        <i:emit/>
94      </get>
95      <get id="string-get">
96        <i:submit request="string-request"
97                  channel="simulator" variable="result" type="string-msg"/>
98        <i:put expression="${result.string}" attribute="string"/>
99        <i:emit/>
100     </get>
101     <set id="integer-set">
102       <!-- Just for test purposes -->
103       <i:create variable="set-data" type="set-msg"/>
104       <i:set variable="set-data" field="test"
              value="${param['integer']}"/>
105     </set>
```

```
106     <periodic id="all-periodic">
107       <start>
108         <i:create variable="period" type="sampling-period"/>
109         <i:set variable="period" field="period"
                value="${param['period']}"/>
110         <i:submit request="all-request" channel="simulator">
111           <i:param name="period" variable="period"/>
112         </i:submit>
113       </start>
114       <stop>
115         <i:create variable="period" type="sampling-period"/>
116         <i:set variable="period" field="period" value="0"/>
117         <i:submit request="all-request" channel="simulator">
118           <i:param name="period" variable="period"/>
119         </i:submit>
120       </stop>
121       <on message="all-msg" variable="result">
122         <i:put expression="${result.integer}" attribute="integer" />
123         <i:put expression="${result.float}" attribute="float" />
124         <i:put expression="${result.string}" attribute="string" />
125         <i:emit />
126       </on>
127     </periodic>
128     <periodic id="multiple-periodic">
129       <start>
130         <i:create variable="period" type="sampling-period"/>
131         <i:set variable="period" field="period"
                value="${param['period']}"/>
132         <i:submit request="integer-request" channel="simulator">
133           <i:param name="period" variable="period"/>
134         </i:submit>
135         <i:submit request="boolean-request" channel="simulator">
136           <i:param name="period" variable="period"/>
```

```
137          </i:submit>
138        </start>
139        <stop>
140          <i:create variable="period" type="sampling-period"/>
141          <i:set variable="period" field="period" value="0"/>
142          <i:submit request="integer-request" channel="simulator">
143            <i:param name="period" variable="period"/>
144          </i:submit>
145          <i:submit request="boolean-request" channel="simulator">
146            <i:param name="period" variable="period"/>
147          </i:submit>
148        </stop>
149        <on message="integer-msg" variable="result" sync="true">
150          <i:put expression="${result.integer}" attribute="integer" />
151          <i:emit />
152        </on>
153        <on message="boolean-msg" variable="result">
154          <i:put expression="${result.boolean}" attribute="boolean" />
155          <i:emit />
156        </on>
157      </periodic>
158      <async id="event-async">
159        <start>
160          <i:create variable="period" type="sampling-period"/>
161          <i:set variable="period" field="period" value="200"/>
162          <i:submit request="event-request" channel="simulator">
163            <i:param name="period" variable="period"/>
164          </i:submit>
165        </start>
166        <on message="event-msg" variable="result">
167          <i:put expression="${result.event}" attribute="event"/>
168          <i:emit />
169        </on>
```

```
170     </async>
171   </operations>
172
173 </device>
```