Chapter 6
SmartSensor Proof of Concept

Abstract This chapter illustrates the use of the SmartSensor infrastructure through the development of an application in the domain of smart buildings. Smart buildings are buildings instrumented with smart devices designed to provide high flexibility of use and the ability to evolve and adapt according to the needs of organizations and human beings, aiming at increasing users comfort and safety and optimizing the operation and managing of several functions inside and outside the building while increasing its energy efficiency. There are plenty of applications within the broad domain of smart buildings, varying from applications to control light, humidity and temperature of rooms to fire and intrusion detection. We choose a parking lot management application to present the main functionalities and potential benefits of SmartSensor. The application consists of a wireless sensor network (WSN) based vehicle detection sub-system connected to the SmartSensor infrastructure. The WSN gathers information on the availability of each parking lot and the SmartSensor infrastructure processes the information and provides a Web interface to guide the driver to the available lots.

Keywords Internet of Things (IoT) · Web of Things (WoT) · REST · Applications for IoT · Mashups · EMML · Parking Lot · Smart buildings

6.1 Overview

This chapter demonstrates the use of the SmartSensor infrastructure through the development of an application in the domain of smart buildings. According to [2], smart buildings are buildings equipped with smart devices designed and constructed to offer great flexibility of use, providing the ability to evolve and adapt according to the needs of organizations and to provide at each moment, the best possible support for their activities. Furthermore, smart buildings must be equipped with systems for automation, computing and communications, which enable, in an integrated and

F. C. Delicato et al., *Middleware Solutions for the Internet of Things*,
SpringerBriefs in Computer Science, DOI: 10.1007/978-1-4471-5481-5_6,
© The Author(s) 2013

consistent way, the effective management of the resources available in the building, boosting increases in productivity, allowing energy savings and offering high levels of comfort and safety to the individuals that work in them.

Examples of smart building applications are: temperature, lighting, air quality and windows (natural ventilation) control; applications that monitor and shutdown unattended devices; security applications to protect personnel (access control) and building properties (anti-theft), parking lot management, detection and management of emergency situations, just to name a few. Such applications often monitor physical variables extracted from the target environment, such as light, vibration, temperature, proximity, presence, chemicals (smoke or gas) and electric voltage.

In order to efficiently manage such applications, the notion of integration arises. Integration is defined as the ability to communicate, collaborate and exchange information between applications to achieve common goals [2]. Examples of advantages of integrating applications are: (i) more efficient use of resources, such as energy, computational, and even human resources, (ii) fast and more coordinated responses to monitored physical events, (iii) the ability to correlate information between applications to optimize the decision process, (iv) decision chaining between integrated applications, i.e., a decision made in a given application may trigger another decision on a different application. As an example, in the occurrence of a fire hazard, a service of detection and management of emergency situations needs to interact with many other services such as: lighting, elevators, parking lot, building access control. These services, whenever informed of the existence of a fire in a particular area of the building can trigger actions such as depressurization and smoke removal of the affected area, pressurization of evacuation areas, automatically disabling elevators and moving the occupied cars to safe floors, prevent access to people in general to areas that may be at risk of being affected by the sinister, allow free exit from sinister places, and block access to building and parking areas that may be at risk.

Many smart buildings services require continuous monitoring of various environmental parameters inside and outside the building using sensors and actuators [1, 2]. Moreover, several services demand interaction between sensing data and information systems that manage the operation of the building. Thus, a key requirement for an efficient monitoring and controlling is that all sensors and actuators are addressable over the network to exchange data with corporative intranet or the Internet. In this context, the use of a WoT infrastructure can bring a set of benefits as the enabler technology to achieve the degree of interoperability among senor instrumented spaces in a smart buildings and an internal or external Web-based network. In the next section, we describe the development of a smart building application using the SmartSensor infrastructure.

6.2 Parking Lot Application

In this section, we describe a smart building application that provides guidance to drivers that need to park a car in one of the available parking lots within a given building. A challenge usually found in applications for managing parking lots is to

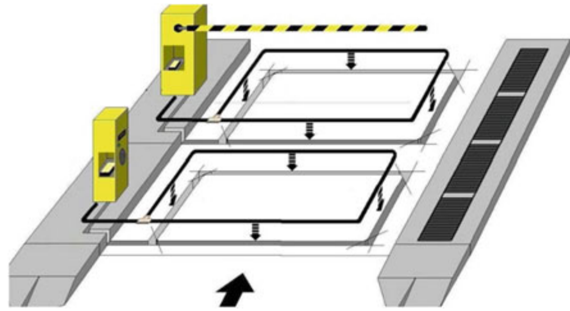

Fig. 6.1 Schematic draw of an inductive loop

effectively detect vehicles. Many solutions use inductive loops (Fig. 6.1) to tackle this issue [3]. However, inductive loops have high costs of both installation and maintenance [4]. In this sense, an easy and cost effective solution option is to use a WSN. Wireless sensors can be easily deployed in existing parking lots without the need for excavation and expensive cable installations required by inductive loops. Moreover, the flexibility to reconfigure sensors already installed, together with the availability of low cost sensors capable of detecting vehicles, make WSN a natural candidate to solve the emerging problems of monitoring and control of parking lots in smart buildings.

Our illustrative application consists of a WSN based vehicle detection sub-system connected to the SmartSensor infrastructure. WSN gathers information on the availability of each parking lot and the SmartSensor infrastructure processes the information and provides a Web interface to guide the driver to the available parking lots.

WSNs have great potential to provide an easy and cost effective solution for the parking lot management application. Its usage along with a WoT solution allows remote and real-time access to the information on the availability of lots besides other useful information, thus increasing the efficiency and manageability of large parking lots, while saving time for the user.

6.2.1 Application Requirements

The Proof of Concept (PoC) parking lot application was developed according to the following requirements:

- The system must provide a list with the location of all parking lots registered in the SmartSensor infrastructure. Such a list must be published as a Web mashup application in order to allow end users to easily locate the nearest parking related to his/her current location.
- The system must provide information about the number of available spaces in each parking lot registered in the system.

- The system must be able to identify the types of available spaces (normal or large) that each parking lot have. Normal spaces fit small vehicles while large spaces fit larger vehicles (trucks).
- The system must be able to provide the location of available spaces within a parking in order to guide the driver to them.
- The system must allow real-time monitoring, via the Internet, of vehicles entrance and exit from a given parking lot.

The UML use case diagram of Fig. 6.2 illustrates the interactions between the end user and the parking lot application according to the requirements described in the previous paragraph.

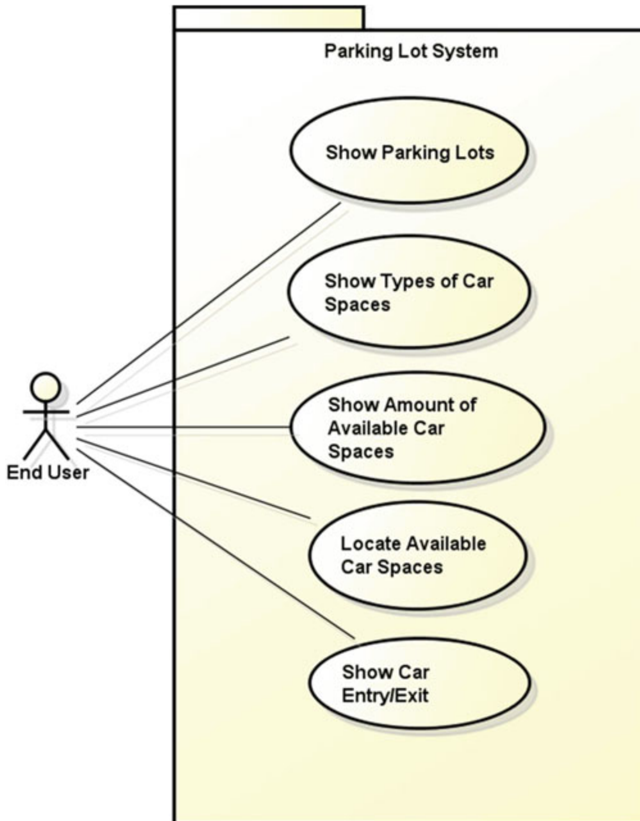

Fig. 6.2 UML use case diagram representing the Parking Lot application

Fig. 6.3 Parking lot image

6.2.2 Environment Setup

The experiment was conducted in three parking lots located at the Center for Mathematical Sciences and Nature (CCMN) in the Federal University of Rio de Janeiro (UFRJ), Brazil. One of these parking lots is illustrated in Fig. 6.3. This parking lot has three rows with 72 places available in each, summing up 216 monitored car spaces.

To detect vehicles and to distinguish them from other objects, such as a person walking through the parking entrance, one pair of sensor nodes endowed with ultrasonic distance detectors were placed at the entrance and exit of each row of the parking lot, as shown in Fig. 6.3. Whenever an object is detected, each pair of sensor nodes sends its collected data to a sink node (Gateway) using a wireless communication channel. A SIM Driver installed in the sink node receives such data, decodes it, and then forwards it to the *SIM Manager component*. Then, the *Manager component* analyses the data and identifies that it must be forwarded to a specific Web service installed in the SIM to further processing. This Web service is responsible for calculating the width of the detected object, and to infer whether it is a car or other type of object. Whenever a car is detected, the Web service accesses the SIM database to update the current number of available car spaces of the parking lot. Regarding the hardware, both the pairs of sensor nodes and the wireless communication module of the sink node consist of Arduino Uno boards endowed with Xbee Shields for wireless communication. The SIM components are installed in a desktop computer and another computer hosts the PEM components. The details of the hardware used for this experiment are found in Sect. 6.2.3.

Figure 6.4 shows a schematic draw of the configuration of the sensor nodes and sink nodes in the parking lot application. A pair of sensor nodes were placed at the

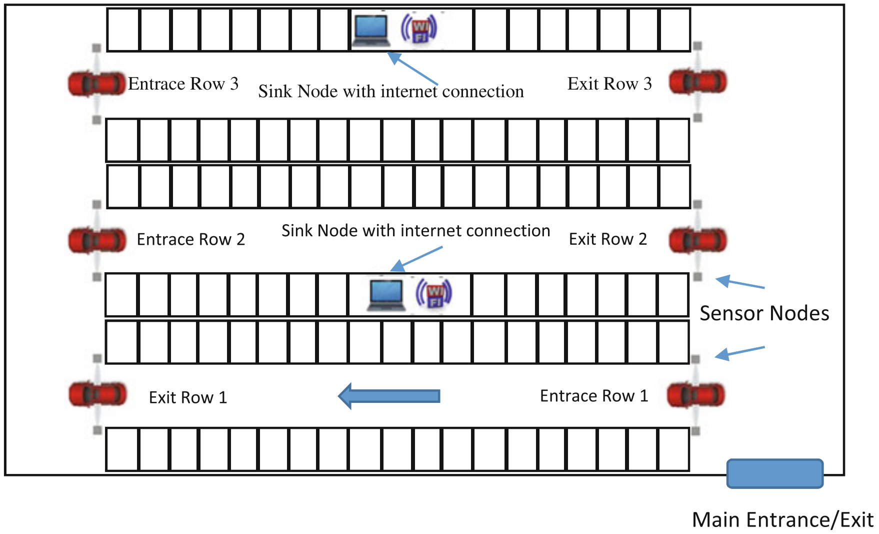

Fig. 6.4 Schematic draw of hardware elements in the parking lot application

entrances and exits of each parking lot row, which is 120 m long and 8 m wide. Two sink nodes were placed in the center of the second and third rows of the parking lot, 60 m away from the entrance of the row. The decision to place the sink node at this location is due to the limitations of radio coverage of the 802.15.4 protocol, implemented by the Zigbee standard [5], which allows data transmission up to a maximum of 100 ft away with direct line-of-sight.

We assume the minimum width of a car as being at least 1.5 m. Therefore, any detected object larger than 1.5 m is considered a car by the application. The pair of sensor nodes at the entrance of each row counts occupied spaces while the par of sensors at the exit of each row counts the vacancy of a space. The sensors of each pair of nodes were placed 8 m apart from each other. To calculate the size of detected objects we use the formula:

$$OS = 8 - DDR - DDL \tag{6.1}$$

where OS is the size of the object, DDR is the distance measured by the sensor node at the right side, and DDL is the distance measured by the sensor node at the left side. Such formula is applied by the SIM Web service on the measurements sent by the sensor nodes.

6.2.3 Hardware Components

The following hardware components were used to instrument each parking lot:

- 14 Arduino Uno boards: 12 used on sensor nodes and 2 used in the sink nodes;
- 14 Arduino Xbee Shield: 12 used on sensor nodes and 2 used in the sink nodes;
- 12 ultrasonic sensors Maxbotix LV-EZ1;
- 12 batteries;
- 2 laptop computers to host the sink nodes.

Arduino Uno (Fig. 6.5) is a microcontroller board based on the ATmega328 chip, which has 14 digital inputs/outputs pins, 6 analog inputs, a 16 MHz crystal oscilator, a USB connection, a power jack, an ICSP header, and a button reset. The Arduino Uno board can be powered via the USB connection or through an external battery connected its power jack.

Arduino Xbee Shield (Fig. 6.6) can be plugged on top of the Arduino Uno to allow it to communicate wirelessly using Zigbee. It is based on the Xbee module from MaxStream [5]. The Xbee module can communicate up to 30 m indoors or 90 m outdoors (with direct line-of-sight).

Ultrasonic Distance Sensor Maxbotix LV-EZ1 (Fig. 6.7) has a frequency of 42 kHz and reading rate of 20 Hz. The LV-EZ1 has virtually no blind spots, detecting objects up to 6.5 m. The closest measured distance is 15 cm, meaning objects closer than this distance are measured as being 15 cm apart. The ultrasonic distance sensor emits a sound signal that travels up to a solid object, like a wall, and back to the source of the sound. To determine the distance of a solid object, the travel time of the echo is calculated.

The laptops were used to host the SIM components responsible for processing the data collected by the base station and transmit them over the Internet to the PEM. The minimum required configuration is:

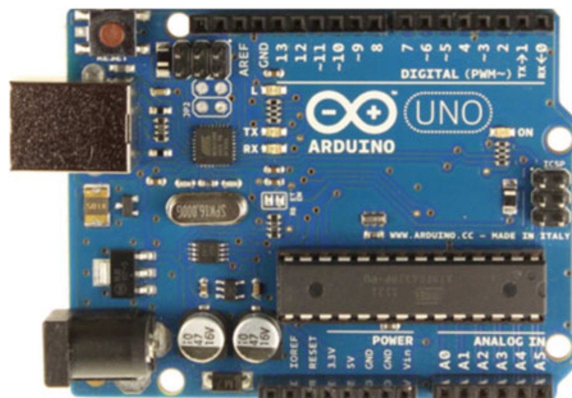

Fig. 6.5 Arduino Uno board

Fig. 6.6 Xbee Shield

Fig. 6.7 Ultrasonic Distance Sensor Maxbotix LV-EZ1

- Hardware:
- 1 gigabyte (GB) of RAM
- 50 megabytes (MB) of available disk space for installation
- Software:
- Linux Operating System

- Database MySql 5.5
- JDK version 1.5 or higher
- Tomcat Server 6

Besides the aforementioned hardware components, the parking lot application uses a server computer responsible for hosting the SmartSensor PEM. The minimum configuration required for this computer to run the experiments is similar to the laptop computers. However, in a real installation this computer needs to be configured according to more specific systems requirements.

6.3 Application Development

The development of the application has three distinct phases. The first phase comprises the hardware configuration of the sensor nodes and the sink node. The second phase comprises the programming of the SIM components required to collect and interpret the signals collected by the WSN ultrasonic sensors. These components are to be installed at sensor nodes and sink nodes. The third phase is the programming and installation of the PEM mashup application built to monitor the parking lots registered in the SmartSensor infrastructure. To mount the ultrasonic sensors on the Arduino Uno boards, it is necessary to follow the following configuration steps (Fig. 6.8):

- To connect the sensor calibration pin to digital pin 13 of the Arduino board.
- To connect the sensor analog output to the Arduino analog pin 0.
- To connect the sensor voltage pin to the Arduino 5 V voltage pin.
- To connect the ground pin to the GND pin of the Arduino.

The configuration of the ultrasonic sensors of the experiment can be visualized in Fig. 6.9.

The first step to program the sensor nodes and configure the Gateway according to the application requirements is to assess the type of the required data delivery model. In general, the data delivery model of WSN applications can be of two kinds (or a combination of both): synchronous or asynchronous. In the synchronous model, network nodes must respond to an application request or should monitor some periodic event. To deal with synchronous events based on Request-Response operations, the SmartSensor provides developers with a REST Web service that is accessed through

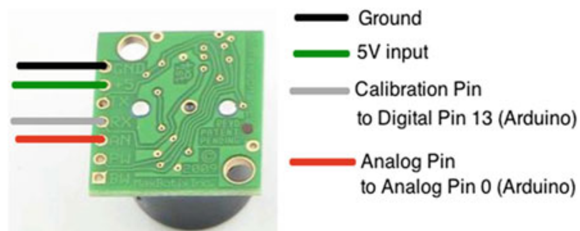

Fig. 6.8 Ultrasonic sensor setup

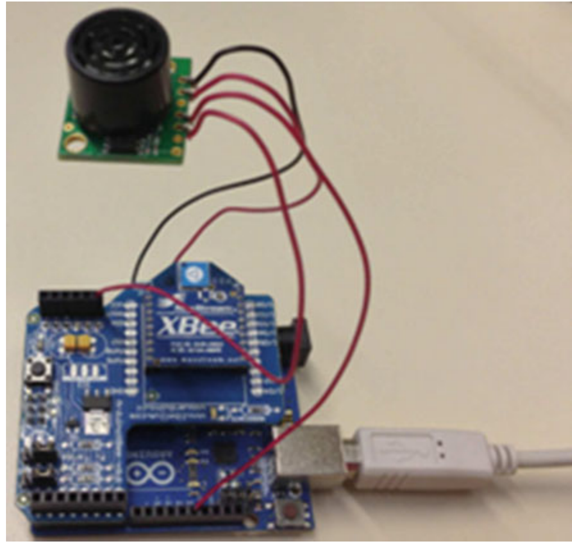

Fig. 6.9 Ultrasonic sensor mounted on the Arduino board

the Gateway and returns the information collected by the sensor. The REST Web service follows the following format:

```
http://{url_mis}/gateway/rest/getdata/{sensor}
```

To handle periodic events the developer must program the sensor node itself to raise such events, i.e., the developer must set the parameters of sensing data rate and sending data rate of the sensor node. For sensor nodes to send data to the gateway, the developer must create an HTTP message using the *createHTTPmsg* method provided by the SmartSensor HTTP library. This method has all the comprised elements of an HTTP message: DHost, Shost, code, method, path, data and error. Where the parameter "DHost" represents the id of the destination node, "Shost" the source node, "code" represents the message type (for example, 2000 for discovery messages and 0 for sending data), "method" represents the HTTP verb (for example G for GET) and "path" represents the type of sensor (for example, 5 for distance), "data" represents the data collected by the sensor, and the parameter "error" sets the error value, if any. To programming the sending of HTTP messages to the Gateway, the developer uses the *sendHTTPmsg* method passing as a parameter a message created by the *createHTTPmsg* method.

SmartSensor also allows processing asynchronous events. These events are unpredictable, and must be configured on the sensor node, creating an HTTP message using the *createHTTPmsg* method and sending it to the Gateway via the *sendHTTPmsg* method.

In the parking lot application, the vehicle detection is programmed as a complex asynchronous event involving a pair of sensor nodes (Master and Slave) and the Gateway. Whenever the Master node detects the presence of an object it sends an

HTTP message to the Gateway informing the occurrence of the event. The Gateway, upon receiving of such message, makes a synchronous request to the Slave node to sense the current distance (object detection). After receiving the Slave response, the Gateway uses both measures to calculate the object size and determine if the detected object matches or not a vehicle. After a vehicle detection, the Gateway performs an update of the number of car parking spaces at the SIM database. This complex event is implemented by a component called "ParkingManager" that must be implemented by the application developer and installed at the Gateway node. The steps involved in the vehicle detection are illustrated in the UML sequence diagram of Fig. 6.10.

After the SIM programming and configuration, the next step in the programming of the parking lot Mashup application that will be installed in the SmartSensor PEM. This step starts with the building of a set of REST Web services that exposes information about the parking lot state. In the PoC application, we developed the following Web services:

- *ListParkingLots: This service provides a list of parking lots registered in infrastructure. The invocation of this service should follow the format:*

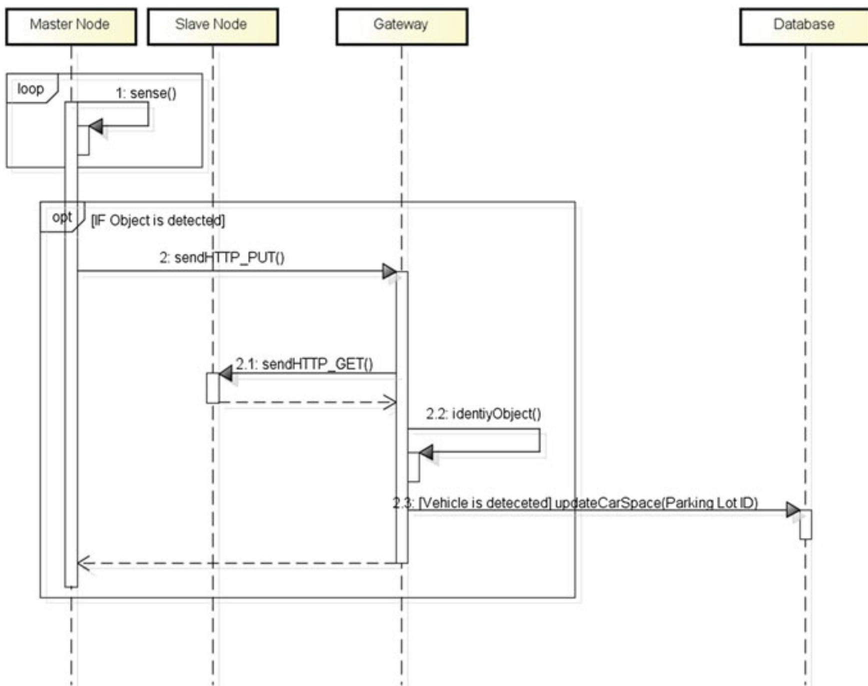

Fig. 6.10 Vehicle detection UML sequence diagram

`http://{PEM_SERVER}:8080/pem-v3.4-emml/ListSimEmml.`

Following an example of the XML file returned as response for this service:


```
▼<Results>
  ▼<Row>
     <LOCATION>R. Lobo Carneiro 470, Rio de Janeiro</LOCATION>
     <GATEWAY>146.164.247.208</GATEWAY>
     <LAST_ACCESS>2013-04-19 17:20:12.0</LAST_ACCESS>
     <FIRST_ACCESS>2013-04-19 15:48:12.0</FIRST_ACCESS>
   </Row>
 </Results>
```


- getParkingLotInfo: This service provides the detailed information of a given parking lot registered in the infrastructure. The invocation of this service should follow the format: `http://{PEM_SVREER}:8080/pem-v3.4-emml/getParkingLotInfo?gatoway={url}`

  where `url` is the address of the Gateway that manages the parking lot. The following code shows an example of the XML file returned as response for this service:


```
▼<Results>
  ▼<Row>
     <AREA_ID>1</AREA_ID>
     <M_SENSOR_I>51</M_SENSOR_I>
     <S_SENSOR_I>61</S_SENSOR_I>
     <M_SENSOR_O>71</M_SENSOR_O>
     <S_SENSOR_O>81</S_SENSOR_O>
     <T_SPACES_N>10</T_SPACES_N>
     <T_SPACES_S>2</T_SPACES_S>
     <C_SPACES_N>6</C_SPACES_N>
     <C_SPACES_S>1</C_SPACES_S>
   </Row>
  ▼<Row>
     <AREA_ID>2</AREA_ID>
     <M_SENSOR_I>52</M_SENSOR_I>
     <S_SENSOR_I>62</S_SENSOR_I>
     <M_SENSOR_O>72</M_SENSOR_O>
     <S_SENSOR_O>82</S_SENSOR_O>
     <T_SPACES_N>10</T_SPACES_N>
     <T_SPACES_S>2</T_SPACES_S>
     <C_SPACES_N>10</C_SPACES_N>
     <C_SPACES_S>2</C_SPACES_S>
   </Row>
 </Results>
```


- getCarSpacesInfo: This service provides the historical information about car spaces of parking lots registered in the infrastructure. The invocation of this service

should follow the format:
`http://{PEM_SERVER}:8080/pem-v3.4-emml/gntCarSpacesInfo?gateway={url}`,

where `url` is the address of the Gateway that manages the parking lot. The following code shows an example of the XML file returned as response for this service:


```
▼<Results>
  ▼<Row>
     <SPACE_ID>1</SPACE_ID>
     <AREA_ID>1</AREA_ID>
     <DATE>2013-03-22 16:00:35</DATE>
     <SPACE_TYPE>N</SPACE_TYPE>
     <IN_OUT>In</IN_OUT>
     <C_SPACES>9</C_SPACES>
   </Row>
  ▼<Row>
     <SPACE_ID>2</SPACE_ID>
     <AREA_ID>1</AREA_ID>
     <DATE>2013-03-22 16:01:02</DATE>
     <SPACE_TYPE>S</SPACE_TYPE>
     <IN_OUT>In</IN_OUT>
     <C_SPACES>1</C_SPACES>
   </Row>
  ▼<Row>
     <SPACE_ID>3</SPACE_ID>
     <AREA_ID>1</AREA_ID>
     <DATE>2013-03-22 16:01:45</DATE>
     <SPACE_TYPE>N</SPACE_TYPE>
     <IN_OUT>In</IN_OUT>
     <C_SPACES>8</C_SPACES>
   </Row>
  ▼<Row>
     <SPACE_ID>4</SPACE_ID>
     <AREA_ID>1</AREA_ID>
     <DATE>2013-03-22 16:01:45</DATE>
     <SPACE_TYPE>N</SPACE_TYPE>
     <IN_OUT>In</IN_OUT>
     <C_SPACES>7</C_SPACES>
   </Row>
  ▼<Row>
     <SPACE_ID>5</SPACE_ID>
     <AREA_ID>1</AREA_ID>
     <DATE>2013-03-22 16:01:45</DATE>
     <SPACE_TYPE>N</SPACE_TYPE>
     <IN_OUT>In</IN_OUT>
     <C_SPACES>6</C_SPACES>
   </Row>
 </Results>
```


After the creation of the PEM services, the next step is to build the Web mashup application that integrate the information provided by the Web services and present

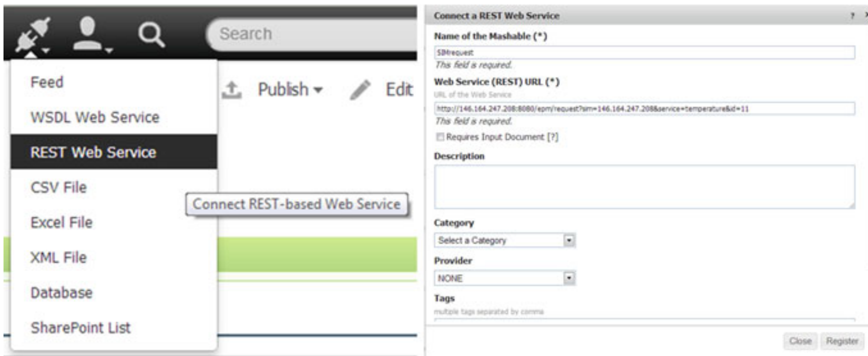

Fig. 6.11 Registering of PEM Web services within presto

this integrated view through a set of graphical user interfaces. Web Mashups can be built using any EMML editor. In this PoC we will show how to create Web Mashups using a popular graphical EMML editor and runtime environment called Presto [6].

First, it is necessary to integrate the PEM Web services created in the last step into the Presto platform. To do so, the Web services must be registered as data source for Mashup applications. This registration is done in the Presto platform through the instantiation of new REST Web Services connections, one for each Web service provided by PEM. Figure 6.11 illustrates the registration process within the Presto platform.

After registering all PEM Web services, we can start building the Web Mashup application. Figure 6.12 is a snapshot of the Presto graphical editor showing the specification of a data flow that processes the information generated by the getCarSpacesInfo Web service, which is represented in the figure by the Presto *Mashable* object ParkingSpace. A *Mashable* object is any object that can be used as data source to create a Mashup application. The data flow specifies that the data received after the invocation of this service should be ordered using the object Sort and forwarded to the object Mashup Output.

Figure 6.13 illustrates the user interface that consumes the output of the EMML Mashup created in Fig. 6.12. This interface shows, in a tabular format and in real time, the entrance and exit of vehicles from a parking lot monitored by the PoC application.

The other functionalities of the PoC application are created using the same process. Figure 6.14 shows the user interface that displays the available car spaces in each lane of a parking lot. This application uses a PEM Web services that queries the database of the SIM responsible for monitoring the parking lot about the current state of its car spaces.

Finally, Fig. 6.15 illustrates the Web mashup application that integrates all the aforementioned user interface fragments into a unified view. The top of the window shows the number of car spaces available in each parking lot lane. At the bottom left is showed the entrance and exit of vehicles, on the right a map indicating the

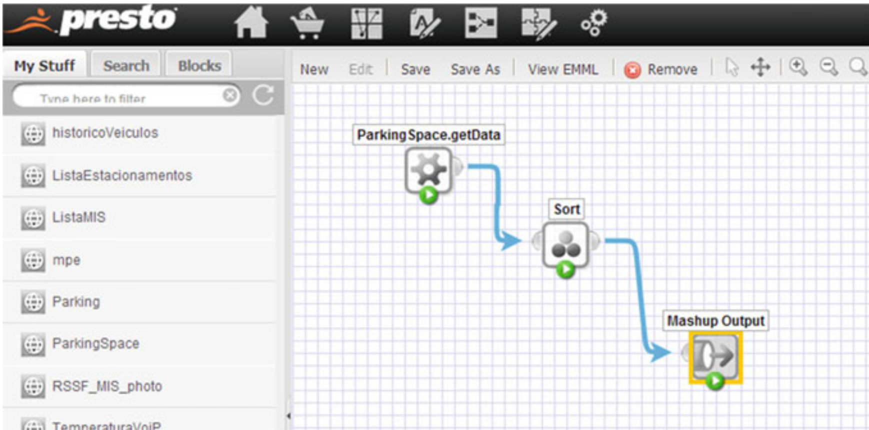

Fig. 6.12 Presto graphical editor

ParkingSpace

Id	Corredor	Tipo De Vaga	Vagas Disponíveis	Data	Entrada/Saída
8	1	N	9	2013-03-22 16:04:55	Out
7	1	N	8	2013-03-22 16:04:46	Out
6	1	N	7	2013-03-22 16:04:30	Out
5	1	N	6	2013-03-22 16:01:45	In
4	1	N	7	2013-03-22 16:01:45	In
3	1	N	8	2013-03-22 16:01:45	In
2	1	S	1	2013-03-22 16:01:02	In
1	1	N	9	2013-03-22 16:00:35	In

Fig. 6.13 User interface for monitoring available car spaces

location of the parking lot, and at the bottom right a table shows general features of the parking Lot, such as total car spaces by type, and current available spaces. Any Web browser connected to the Internet can access this information.

Fig. 6.14 User interface showing the available spaces in a parking lot

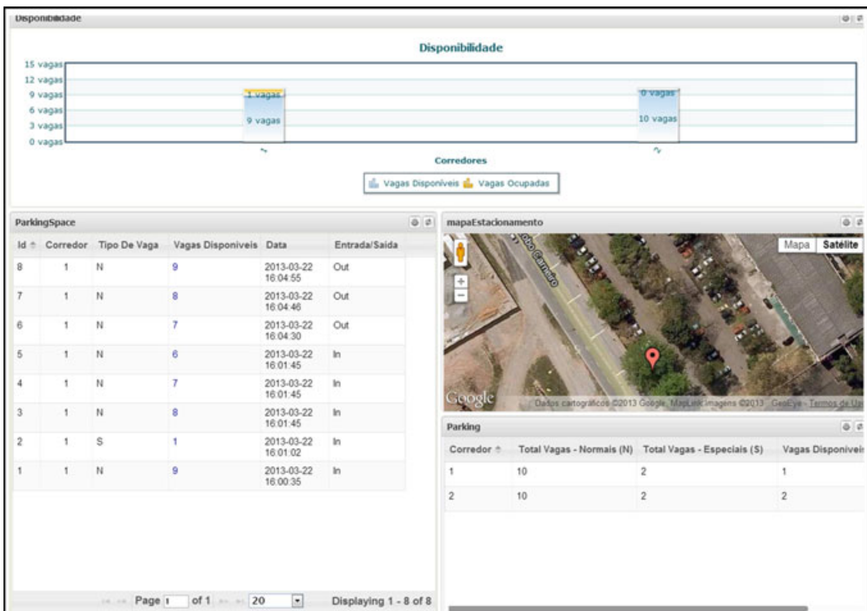

Fig. 6.15 Parking Lot Web mashup application

References

1. Liu, M., Mihaylov, S., Bao, Z., Jacob, M., Ives, Z., Loo, B., et al. (2010, September). SmartCIS: Integrating digital and physical environments. *SIGMOD Record*, *39*, 48–53.
2. Nunes, R. J. C. (1995, November). *Integracão de Serviços para Edifcios Inteligentes*. (in Portuguese) PhD thesis, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal.
3. Chinrungrueng, J., Sunantachaikul, U., Triamlumlerd, S. (2007). Smart parking: An application of optical wireless sensor network. *Proceedings of the 2007 International Symposium on Applications and the Internet Workshops (SAINTW07)*.
4. Vijay Kumar, P., Siddarth, T. S. (2010). A prototype parking system using wireless sensor networks. *International Joint Journal of Power, Control and Signal Processing* (*IJJCET2010*), *1*(4), 78–82, www.ijjcet.com
5. Zigbee Protocol, http://www.sparkfun.com/datasheets/Wireless/Zigbee/XBee-Manual.pdf
6. Presto Platform, Retrieved May 2013, from http://prestocloud.jackbe.com/presto