Chapter 4
The Sensor Integration Module (SIM)

Abstract The SmartSensor architecture encompasses three main software modules: (i) the *Sensor Integration Module* (SIM), (ii) the *Programming and Execution Module* (PEM), and (iii) the *Web 3.0 Integration Module* (WIM). In this Chapter we detail the SIM logical and physical components as well as their operation. In the SmartSensor infrastructure a set of wireless sensor networks (WSN) is connected to the Web through one gateway node, that exposes to client applications the sensing data produced by the networks as RESTful Web resources. The Sensor Integration Module (SIM) is responsible for providing the RESTful interface to access the resources of a given WSN. Its components receive application requests describing their desired sensing data, translate HTTP messages to and from the several sensor specific formats and protocols, coordinate the functions needed to meet the received sensing tasks and manage the different communication models required to produce and deliver the data back to the requesting applications.

Keywords Web of Things (WoT) · REST · HTTP · XML · JSON · Wireless sensor networks · Integrating WSN · Restful services

4.1 Overview

As previously stated, the SmartSensor project considers a system consisting of a set of wireless sensor networks with technologies/platforms possibly distinct, connected to the Web through one gateway node, and a set of client applications. The WSNs are exposed and their data accessed by applications as Web resources, using the concept of RESTful services. The access to the resources provided by a specific WSN is realized through the Sensor Integration Module (SIM). The following Sections present the SIM logical and physical architecture, describing how its software components are deployed in each type of node that composes the SmartSensor infrastructure. As we discussed, SIM components are deployed in sensor nodes and in gateway nodes.

The current implementation of SmartSensor considers WSN nodes from MEMSIC,[1] Arduino,[2] and SUN SPOT platforms.[3]

4.2 The SIM Logical Architecture

The UML diagrams of Figs. 4.1 and 4.2 illustrate the main components of the SIM logical architecture. The deployment diagram of Fig. 4.1 provides an overview of components for each type of physical node (sensor and gateway) considered in SmartSensor. The UML class diagram in Fig. 4.2 details the classes and subclasses that compose the gateway Communication Component.

4.2.1 Gateway Components

As depicted in the diagram of Fig. 4.1, the logical architecture of the gateway node is organized into five software components, described below.

4.2.1.1 Web Interface

This component is the ultimate responsible for providing a uniform Web interface to access the WSN *as a service*. It enables that services provided by the sensor nodes of a

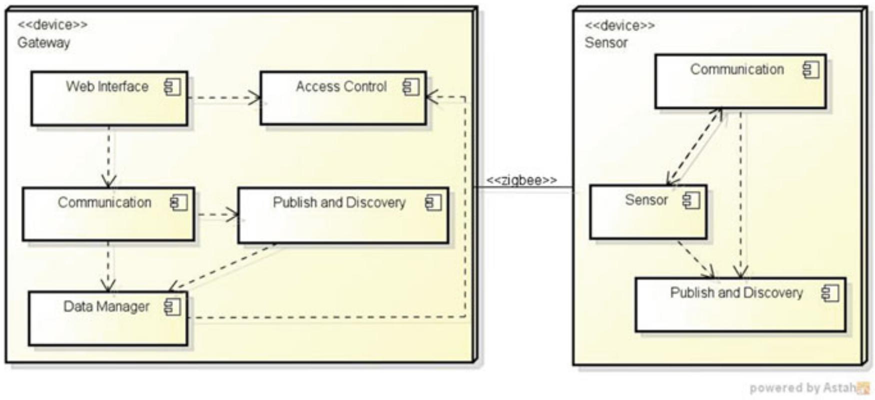

Fig. 4.1 SIM components deployed in the gateway node and in the sensor nodes

[1] http://www.memsic.com/

[2] http://www.arduino.cc/

[3] http://www.sunspotworld.com/

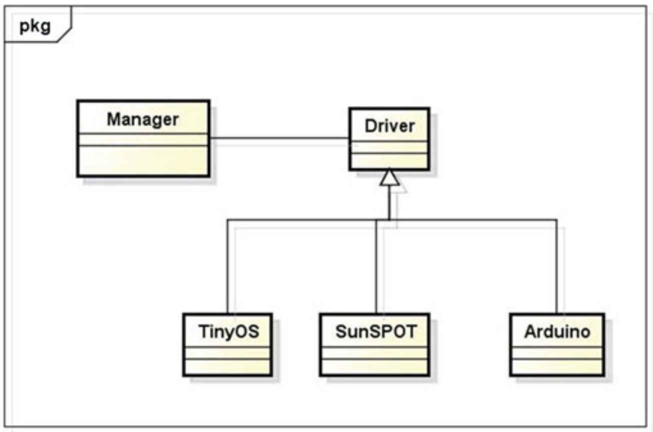

Fig. 4.2 Classes of the communication component

WSN connected to the SmartSensor infrastructure be available in the Web as RESTful resources. Considering the traditional operation phases of a WSN, this component is responsible for phase (ii) submission of application requests (see Sect. 3.2). The main class of this component is called *SIM_Web_Service* and it simply consists of a traditional REST-based Web Service installed in a Web server. *SIM_Web_Service class* handles all the HTTP messages exchanged between client applications and the WSN connected through the respective gateway node.

In order to achieve its goals, the *SIM_Web_Service class* directly interacts with the classes of the *Communication component*. HTTP requests received from applications are processed in the Web server 'as any other request for Web resources. A request message identifies through the URI path (i) a given WSN resource (accessed via the respective driver type), (ii) a specific device (considered as a sub resource of the driver) and (iii) some functionality provided by the device (considered sub resource of the device). Thus, the path of an HTTP request is initially used to identify the type of driver from the device whose service is being requested, then to identify a particular device (if desired) and finally the service (type of sensing data) provided by this device. For example, in the path /spotApi/spot-0f40/temperature, the first part "/spotApi" identifies the driver for this type of device (indicating that it is a Sun SPOT platform sensor). The second part "/device-0f40" identifies the specific node (SPOT), where "0f40" is the last four digits of the SPOT MAC address. Finally, the part "/temperature" is used to identify the temperature sensing unit of the respective SPOT. After analysing the content (body and header) of the HTTP request message, the description of the required sensing task needs to be extracted from the message and forwarded to the sensor nodes able to attend such request. The *Manager class* of the *Communication component* is responsible for determining the nodes that are able to perform a received sensing task. Therefore, after processing an HTTP request message, the *SIM_Web_Service class* reports its content to such component.

Likewise, results (sensor data) provided by the sensor nodes in response to the received requests are sent back to the requesting applications as HTTP reply messages via the *SIM_Web_Service class*. When data produced by the sensor nodes in a WSN is to be sent to a client in response to a given HTTP request, such data is mapped into a REST compliant representation. Possible formats are HTML, XML and JSON. This mapping is responsibility of the *Driver class* of the *Communication component*.

4.2.1.2 Communication

Considering the WSN operation phases, this component includes the several classes responsible for performing the phase (iii) data collection and delivery (see Sect. 3.2).

From the HTTP request messages received and processed by the *SIM_Web_Service class*, this component manages and distributes sensing tasks to the respective sensor nodes, collects the received results and forwards them back to the Web server so that they are properly delivered to the requesting application.

The Manager Class. The *Manager class* of the *Communication component* directly interacts with the *SIM_Web_Service class* and determine, based on the analysis of the incoming messages content and by querying the database maintained at the gateway, which nodes are able to meet the received request (Fig. 4.3). The main parameters used to perform the matching between a requested sensing task and the nodes in a given WSN that are able to execute the task are (i) types of environmental variables to be monitored (depend on the sensing units available in the node); (ii) geographical location of the node; and optionally (iii) quality of service (QoS)

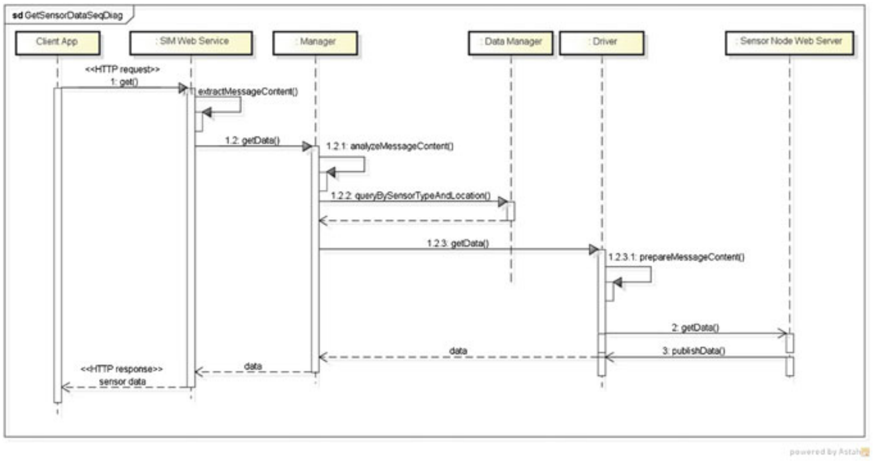

Fig. 4.3 Simplified view of the interactions for the submission of application request messages; message processing; and sensor data delivery

requirements as, for instance, the minimum data accuracy provided by the node, the maximum delay delivered by the network, the maximum lifetime of the node, among others. Upon determining which sensor nodes are used to execute the required sensing task, the *Manager class* is able to know the respective sensor platform(s) to be used to meet the request. Thus, the request message is forwarded to the respective *Driver class* to be translated to the proper data format. After the required sensing task is performed by the WSN and the required sensing data is collected by the nodes, the respective *Driver class* sends to the *Manager class* the translated reply messages directly received from the tasked sensor nodes. The Manager then forwards the message to the *Web Interface component* so that the results of the HTTP request are presented/delivered to the user/client application. The other functionality of the *Manager class* is to determine the type of communication to be used (synchronous or asynchronous) in the message exchange between the gateway and the WSN. Such type is defined from the data delivery model required by the client application.

According to the data delivery model, WSNs can be typically classified in three types: periodic, event-driven and initiated-by-the-observer (or simply request-reply model). In the periodic model, sensor nodes sense and send their collected data continuously, at a predefined rate. In the event-driven model, sensors continuously sense the monitored environment but report information only if an event of interest for the application occurs. In the request-reply model, sensor nodes report their data in response to a synchronous query issued by the application. In this last case, the application is interested in getting a snapshot of values of the monitored phenomenon.

For event-based applications, asynchronous communication is required, for instance, based on the Publish-Subscribe model. The current architecture of SmartSensor does not support this model. With such model, a client application registers for events of interest only once and receives new sensor measurements upon the occurrence of an event. The HTTP protocol typically operates in the pull mode, where clients send a request message whenever they need a resource from a Web server. HTTP does not natively provide an event notification mechanism (push mode). A usual way of implementing the push mode would be to repeatedly send an HTTP request message (for instance containing a conditional GET operation) describing the event of interest; whenever the event occurs the reply message body will include the event description; otherwise the message body will be empty. This implementation based on sending repeated requests makes costly the communication for this type of data delivery model. To overcome such drawback, a possible solution would be to modify the original HTTP protocol implementation. One example of such a solution is the TinyREST protocol [5], proposed as part of a joint R&D project between Samsung Advanced Institute of Technology and Fraunhofer FOKUS. TinyREST is a protocol specific to the TinyOS sensor platform that was built based on the REST architecture and principles. The TinyREST implementation provides the clients with the ability to issue HTTP-like messages to accessing MICA [3] motes in a WSN. Besides the standard POST and GET HTTP operations, TinyREST includes a SUBSCRIBE request message. By issuing a SUBSCRIBE message, clients are able to register their interest to specific services provided by sensors/actuators, besides defining personalized parameters depending on the clients needs. Each subscribed client will automatically be notified with a NOTIFY message whenever a desired event has been detected (e.g. a temperature value passing a specified threshold).

Although providing an efficient way for handling event-based applications and asynchronous communication in a WoT connected WSN, the solution offered by TinyREST actually changes the standard HTTP API and implementation. For a WoT solution that needs to be fully compliant to the REST principles, as is the goal of the SmartSensor framework, this is not a suitable option.

Other options involve introducing a third party component to mediate HTTP messages sent by applications to the gateway. An example of such a solution is the Pubsubhubbub protocol.[4] This protocol enables the communication between client and server using a Publish-Subscribe model by employing a component, called Hub, that registers clients (Subscribers) interested in receiving events (about sensor generated data), gets new data provided by the server (the gateway, acting as a Publisher), and deliver data to the respective clients. The SmartSensor designers consider that handling asynchronous communication in Web-enabled WSNs is still an open issue that requires further investigations to be implemented in an efficient and interoperable way.

A periodic data delivery model is implemented in SmartSensor by the submission of a sensing task that describes the desired data type and the frequency of data delivery (data sensing/sending rate). It requires that the user (or application) access the SIM to check the latest data collected by the network. The SIM database is periodically updated with the latest data sent by the sensors, with the frequency previously configured in the nodes. To access the collected data, the user must access the URI: GATEWAY/gateway/rest/getdata/data type.

Such request will return all sensor data of the required type that were collected and stored in the SIM database until the moment of the request. If the user is accessing SmartSensor via PEM, there is the option to automatically refresh the application, which can be configured according to the required frequency, avoiding the need for the user need to resubmit the request or manually update the HTML page where the data is being displayed.

The Driver Class. Another important class of the *Communication component* is the *Driver class*, a super-class that represents the interaction with the sensors from each specific WSN platform to be integrated in the SmartSensor infrastructure. Drivers translate messages and commands to the specific language/protocols of the WSN and vice-versa. This class is extended by subclasses for each sensor platform. As we have already stated, SmartSensor currently provides drivers for the Arduino,[5] SUN SPOT[6] and TinyOS[7] sensor platforms.

The main operations provided by the *Driver class*, regardless of the sensor platform used are described as follows. The advertiseService operation is responsible for handling the advertisement messages (RequestAdvertiseMessage) sent by the gate-

[4] http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html

[5] http://www.arduino.cc/

[6] http://www.sunspotworld.com/

[7] http://www.tinyos.net/

way for devices that interact with the respective driver. These messages are generated by classes of the *Publish and Discovery component* and should be sent to the sensors, which respond with a message (AdvertiseMessage) advertising their services, residual energy, among other relevant information. The getData operation is used to task the sensor nodes to collect the sensing data as requested by the client application, according to the desired data delivered model. The publishData operation is responsible for receiving data messages sent by the sensors (containing the collected sensing data).

4.2.1.3 Publish and Discovery

There are two levels of service discovery in a WSN: internal and external (phase (i) of the networks operation, as described in the Chap. 3), and both are implemented by classes from the *Publish and Discovery component*.

The internal discovery enables that sink/gateway nodes know the capabilities of all sensor nodes that compose a given WSN connected to the gateway. In order to implement this feature, in the SmarSensor infrastructure a special message, called AdvertiseMessage, was defined to allow sensor nodes to advertise their capabilities. Such message include the node (local) identifier, a timestamp, the types of sensing units available in the node, geographic location, residual energy, maximum data accuracy/precision provided; supported data rates, supported aggregation functions and supported duty cycles. Advertising messages are sent by a sensor node (i) at the node initialization (upon the network deployment in the target area), (ii) when a new sensor joins a pre-deployed network, and (iii) from time to time, either as a keep alive message sent with a predefined periodicity or upon request by the gateway (via a RequestAdvertiseMessage). Such periodic sent of advertising messages is required given the dynamic nature of the WSN environment, where sensors may be damaged, moved, have their energy depleted, thus no longer participating from the network infrastructure. In SmartSensor, if a connected device does not respond to three consecutive RequestAdvertiseMessage sent by a gateway, such device is considered unreachable and should be removed from the list of devices maintained in the database. AdvertiseMessage messages are disseminated throughout the network by using the communication/routing protocols available at the nodes, until they reach the gateway node. In the gateway, the content of such messages is extracted and stored in a database containing data for the respective WSN connected to the gateway. Gateways are organized in a logical hierarchy and interact among themselves in order to exchange data from their respective WSNs. PEM components execute in the gateways positioned in the highest level of the hierarchy. While gateways at the lower levels only keep information on their respective WSNs, the higher level gateways keep a database with updated information on all networks connected to the SmartSensor infrastructure.

The external service discovery is used by client applications to discover which WSNs provide the services they require, and how to access such services. This is a traditional phase of service discovery according to the Web Services technologies.

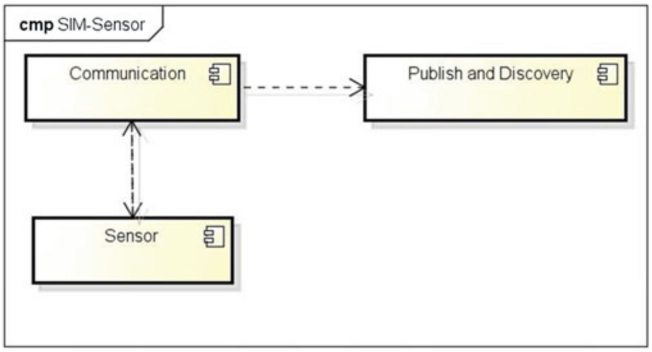

Fig. 4.4 Logical components of sensor nodes

For REST-based Web services, the discovery and navigation through available resources are performed by using URIs. SmartSensor infrastructure provides a REST-based discovery service available through a URI exposed by the Programming and Execution Module (PEM).

4.2.1.4 Access Control

This component includes classes with basic functions (authentication and authorization) for managing the access constraints to the services provided by a given WSN (accessed through a gateway node). Policies are enforced over resource publication and sensing task allocation according to criteria set by network administrators.

4.2.1.5 Data Manager

This component is responsible for storing data in a local database maintained by the gateway. The classes of this component manage the reading and writing operations of the tables responsible for storing sensing data as well as all the information (metadata) about sensing capabilities offered by each network node in the WSN connected to the respective gateway.

4.2.2 Sensor Node Components

Figure 4.4 shows the SIM main software components that should be deployed in the sensor nodes. The components are described as follows.

4.2.2.1 Communication

As we previously mentioned, WoT employs REST principles to expose the services of smart devices available on the Web by using two different approaches. In the first approach, an embedded HTTP server is deployed directly on the devices and the functionality of these devices is provided as RESTful resources. The second approach is adopted whenever a device does not have hardware resources enough to run an embedded server, or when it is not necessary that such device is directly accessed via Web. For these cases, another, more powerful device can be used as a bridge to expose the services provided by the constrained device via a RESTful interface. Such device consists of a WoT gateway. In the SmartSensor project both approaches were implemented. However, as mentioned, independently of either having a server embedded in the sensors or not, the gateway is always used for mediating the interaction of WSNs with the Internet (for the purposes of converting the adopted protocol stacks).

For the first approach, an embedded Web server is directly implemented on each sensor node making it an autonomous and Web-enabled device. The use of servers embedded in physical objects enables the functionality of these objects to be available as Web resources. However, the technologies used in the creation of traditional Web services are not designed to be used on devices that are severe restricted in resources and battery powered (eg, wireless sensors) [4]. Therefore, so that Web servers are used in embedded devices, they must meet a number of requirements. In Ref. [4] a set of requirements and standards for the implementation of embedded servers were presented. An example of a requirement to be met in a standardized way is the compression of HTTP protocol messages [1]. For the definition of a generic architecture for embedded servers, the SmartSensor project followed a bottom-up approach, in which such an architecture was derived from an existing implementation of a embedded server deployed in a specific sensor platform, the SUN Spot. The implementation used as a reference for the SmartSensor design is described in the WebOfThings project.[8] From the analysis of the components of this existing architecture, a platform-independent generalization was performed and adopted in the SmartSensor logical architecture to guide the possible implementation of a server embedded in other sensor platforms.

The embedded Web server is basically a very lean version of an HTTP server, capable of handling HTTP request messages and generating reply messages server. Thus, it natively supports the four main operations of the HTTP protocol (GET, POST, PUT, DELETE, i.e the verbs of REST). In this case, the *Communication component* in the sensor nodes encompasses the typical classes of an HTTP engine, including a request dispatcher and a response builder [2].

For the second approach, the *Communication component* includes the classes and interfaces native for each sensor platform, which are responsible for the communication tasks. Such classes should participate in the completion of three tasks: (i) sending messages advertising the capabilities of the device; (ii) receiving

[8] http://www.webofthings.org/projects/

messages requesting for a given sensing task/data, and (iii) sending reply messages in response to request messages (after the required sensing tasks have been performed).

4.2.2.2 Sensor

This component includes classes responsible for keeping the current state of the sensor nodes, both regarding the available resources, such as residual energy, and the acquired sensing data. As the data collected by the sensing units are not always immediately transmitted (depending on the data delivery model required by the application and also on the adopted data aggregation intervals), the classes of this component shall keep the data in the node memory until they are processed and sent through the network towards the gateway.

4.2.2.3 Publish and Discovery

The classes of this component are responsible for implementing the internal discovery service. Therefore, a class is required to create messages advertising the node features and send these messages whenever required. Classes of the Publish-Discovery and the *Sensor components* directly depend on the low-level primitives provided by the sensor operation system.

4.2.3 The SIM Physical Architecture

The logical architecture previously described for SIM was instantiated on a gateway node implemented in Java and on sensor nodes from three different technologies: Mica platform/TinyOS, Arduino and Sun Spot. In the next subsections we describe the gateway physical architecture and the components for the MICA/TinyOS platform. Description of the components implemented for Arduino and Sun Spot platforms are outside the scope of the Book.

4.2.3.1 WoT Gateway

Despite the REST principles are suitable for the integration of physical devices to the WoT, such devices do not always have sufficient computational resources to support an embedded server. Therefore, the direct integration of real-world devices with the Web is still a complex task, especially in cases of extremely limited resources devices such as the sensor nodes in a WSN. In such cases, a different strategy for the integration should be adopted, based on the utilization of an intermediate device, Smart Gateway or WoT gateway. Smart Gateways have two basic functions: to expose a RESTful interface via URIs that identify and provide access to physical objects (smart devices) and their resources, and to realize the communication with physical objects using their provided APIs. In other words, the Smart Gateway acts as a bridge between the Web and smart devices, by providing a RESTful Web interface to access resources and sub resources provided by these devices and communicate with them through their specific APIs. The gateway node plays the role of an interface between client applications and WSNs connected to the WoT, serving as the entry point for the submission of application requests and as a concentrator for data sent by the sensor nodes.

In the SmartSensor architecture, a gateway node is a synonymous of a sink node, and its functionalities are partially implemented in a computer (a PC running Debian GNU/Linux i386 in the project) and partly on wireless communication modules that are dependent on the different radio technologies used in WSNs. All the WSN platforms used in the SmartSensor project adopt variations of the ZigBee protocol [3]; therefore the sink/gateway wireless module implements this protocol to enable the communication with sensors.

Each Smart Gateway has an IP address, runs an HTTP server and includes several drivers, each one responsible for translating to/from proprietary protocols of the different WSN technologies connected to the infrastructure. Thus, all Web requests sent to a sensor node through the provided RESTful API are mapped by the gateway to a request in the proprietary WSN API and transmitted to the respective node by using the communication protocol understood by the device (for example, the Zigbee protocol).

The classes and components described for the SIM logical architecture were implemented in the Java programing language. For the Gateway Web Server, the Apache Tomcat version 6.0.33 was used and Apache Derby relational database was adopted as the Gateway Database. The *Data Manager component* is responsible for data storage and management in the Gateway Database and its mains class is the *DataDB class*. DataDB is a typical persistency class, mediating all the read and write operations performed in the two main tables kept in the gateway. The Data_Read table is responsible for the storage of the sensor generated data, while the Services table contains the list of capabilities offered by each node in a given WSN.

4.2.3.2 MICA/TinyOS Sensor Plataforms

MICA motes are the category of sensor nodes manufactured by MEMSIC (formerly Crossbow). MEMSIC technology for WSN platforms is based on the TinyOS operating system and programs to be deployed in the nodes are written in nesC language. As specified in the MIS logical architecture, a sensor node must have three basic functional blocks to be integrated into the SmartSensor infrastructure: Communication, Publication, and Sensor. TinyOS adopts a component-based and event-driven programing model, and nesC is a language derived from C, so it does not natively incorporate concepts of object-oriented programing. The main units of programming in TinyOS environment are components and interfaces. Therefore, in order to implement the functionalities of the three logical blocks defined for the sensors three

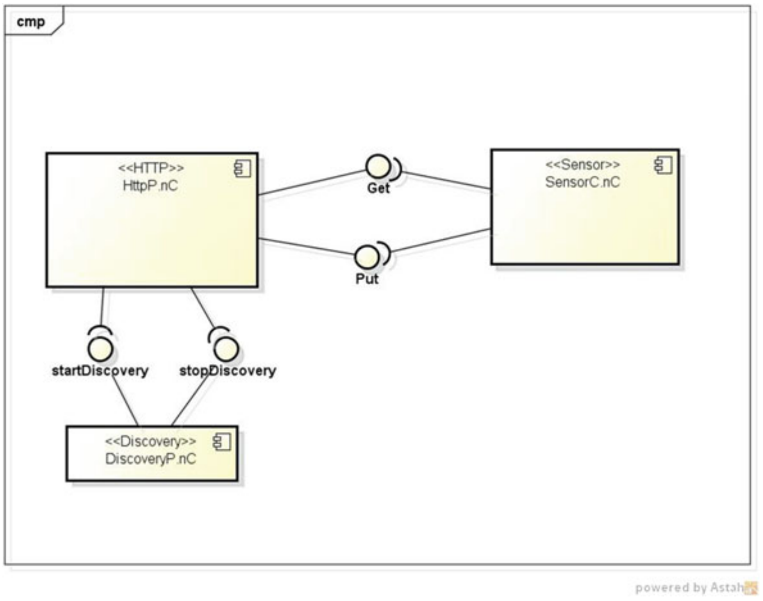

Fig. 4.5 Components e interfaces of sensor nodes in the TinyOS platform

major components and their respective interfaces were created. Such components and interfaces are illustrated in the diagrams of Fig. 4.5, and briefly described below.

In the version implemented for the Mica/TinyOS platform, the approach adopted for the integration in the WoT was based in the implementation of an HTTP server embedded in the sensor nodes. Therefore, for such WSN platform, the *Communication component* includes classes responsible for receiving and processing HTTP request messages, and then for composing and sending the respective HTTP reply messages. The features of the *Sensor component* are realized by software components already existing in the sensor platform; it was not necessary to implement them. However, the implementation of the *Publish-Discovery classes* was hampered by the available node interfaces. The access to the node state information on Mica platform is restricted to the sensed data, and there is no API to report, for example, the residual energy of the sensor. Information such as the maximum precision provided by a given sensing unit comes preconfigured from the factory, and there is no native method to get/set such an information. Data such as the node geographic location is only available either if the node is endowed with a GPS unit or if some algorithm for node location is employed. Therefore, in the current version of the SmartSensor infrastructure all the relevant metadata for sensor nodes from the Mica/TinyOS platform was statically configured as parameters in the advertising messages sent by the nodes.

The main software components implemented for the Mica/TinyOS sensor nodes are showed in Fig. 4.5 and briefly described below:

- *SensorC*: this component is deployed in each sensor node to implement the communication based on the interfaces provided by HttpP.
- *HttpP*: this component implements the HTTP protocol API, providing RESTful interfaces for communication with the gateway node (and also between the sensor nodes themselves).
- *DiscoveryP*: this is the component responsible for the (internal) publication and discovery of the capabilities of sensor nodes.

In addition to the components of the sensor node, a component is necessary to connect the WSN (based on TinyOS/nesC) to the gateway (in Java). This component is baseC, implemented in nesC, and responsible for making the connection with the Gateway Web Server through a serial communication interface.

4.2.3.3 Operation

As previously mentioned, a WSN integrated to the WoT works according to three phases: (i) internal and external service discovery, (ii) submission of sensing tasks, (iii) data collection and delivery. Except for the external discovery phase, which is totally the responsibility of the gateway, the other phases are implemented by the sensor node components previously described. During the internal service discovery phase an HTTP PUT message is used to advertise the sensing capabilities of each node to the gateway, thus respecting the RESTful principles to maintain a uniform interface for accessing all data (and metadata) from the connected sensors.

Phases (ii) and (iii) of the network operation are illustrated in the UML activity diagram of Fig. 4.6. In the diagram, the Client swimming lane represents the client side of an HTTP-based interaction with a Web Server. The Gateway Web Server remains listening in a well-known port and waiting to receive a request from client applications, which may be requests for changing some parameters of the sensor (PUT operation) or requests for some monitoring data collected by the sensor (GET operation). In both cases, the received request messages are sent for analysis and subsequent forwarding to the destination sensor node(s). Upon arriving at the gateway, the request message header is analysed, and the following cases are possible: if the message is addressed to the sink node itself, it examines if it is either a Get or Put message; otherwise, the error message 405 is returned to the client. If the message is addressed to the client, it is not forwarded to the WSN, being processed within the gateway. If the message is directed neither to the client nor to the Sink node, a 404 error message is returned to the client. Otherwise, the message is redirected to the specified sensor, group of sensors or broadcasted in the whole network.

Upon the arrival of a message in a sensor node, the message header is analysed, and the following options are possible: if the message is addressed to the sensor itself, it checks whether it is a Get or Put message, if is not either type a 405 error

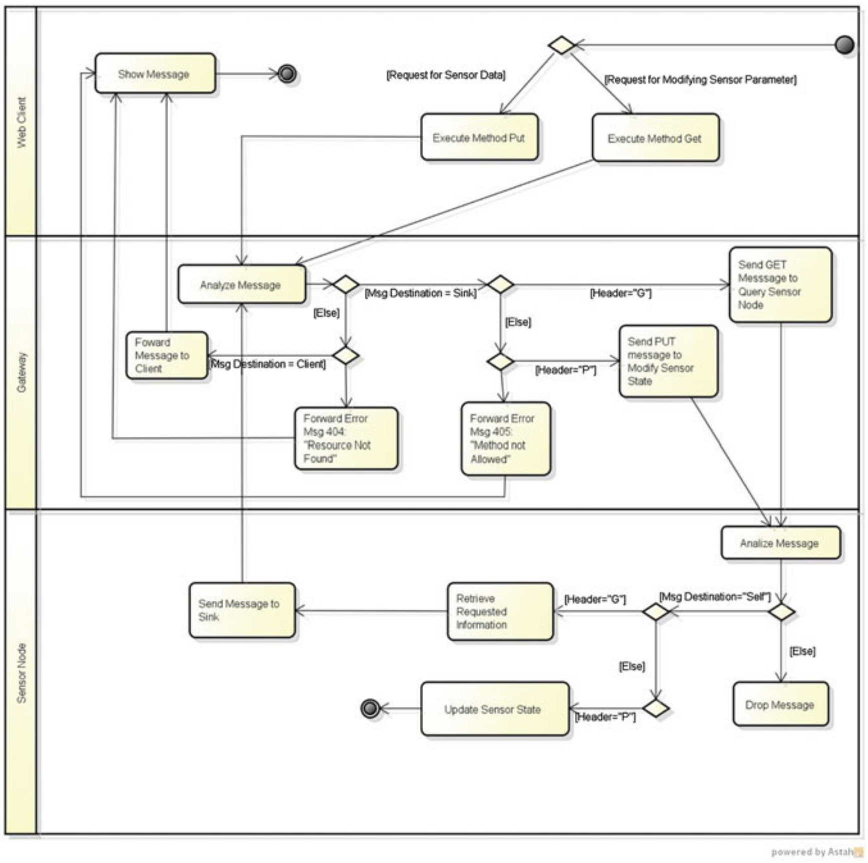

Fig. 4.6 Realization of phases for submission of sensing tasks, data collection and delivery

message is returned. If it is a Get message, the sensor produces a reply message with a copy of the requested information and returns it to the Sink node, which will forward it to the client. If it is a Put message, the sensor will change the current configuration parameter (for example, data sensing or sending rate) according to the values contained in the message.

References

1. Chopan—Compressed HTTP Over PANs (draft-frank-6lowpan-chopan-00). http://w3-org.9356.n7.nabble.com/Chopan-Compressed-HTTP-Over-PANs-draft-frank-6lowpan-chopan-00-td104660.html. Last access April 2013
2. Guinard, D., Trifa, V., Pham, T., & Liechti, O. (2009). Towards physical mashups in the web of things. *Proceedings of IEEE Sixth International Conference on Networked Sensing Systems*. Pittsburgh, USA

3. MEMSIC solutions. http://www.memsic.com/products/wireless-sensor-networks.html
4. Shelby, Z. (2010). Embedded web services. *IEEE Em Wireless Communications*, *17*, 52–57.
5. Luckenbach, T., Gober, P., Arbanowski, S., Kotsopoulos, A., & Kim, K. (2005). TinyREST—a protocol for integrating sensor networks into the internet. *Proceedings of REALWSN 2005*