A Framework for Integrating Wireless Sensor Networks and the Internet

Jeisa Domingues, Antonio Damaso, Rilter Nascimento and Nelson Rosa

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/50118

1. Introduction

The integration of Wireless Sensor Networks and the Internet is growing in importance as WSNs have been employed in a great variety of applications which need a common means to share data over the Internet. In the recent years, many solutions have been proposed to provide that integration. The simplest and most popular one is the gateway-based approach [1], which basically converts between protocol stacks and logical address formats used in both networks. In the overlay-based approach [2], some sensor nodes may implement the TCP/IP protocols or some hosts may implement WSN protocols, being the reachable nodes and acting like gateways in their respective network. Another solutions use the TCP/IP suite as the communication protocols for sensor nodes [3]. These three approaches focus on accessing the network nodes through their logical addresses, which has several problems, such as the different addressing and routing scheme of both networks. Besides, employing TCP/IP in the sensor nodes raises specific issues, like the header overhead of those protocols which is very large for small packets, and the end-to-end retransmissions used by TCP which consume energy at every hop of the path.

Mobile agents have also been used as an approach to dynamically access the WSN from the Internet [4], answering queries while migrating through the sensor nodes. Another solution adopts service-oriented middleware to integrate WSNs and the Internet, converting sensor nodes into service providers for the Internet hosts [5]. Additionally, Web service approaches have been presented [6], some of them converting all the WSN into a single web service, and others allowing sensor nodes to offer their data through Web services that can be accessed from the Internet. The main limitation of these approaches is that sensor nodes are considered only as service providers, not consumers.

In order to overcome the aforementioned problems, this chapter introduces a framework for integrating WSNs and the Internet by allowing the interoperability of their services. Our approach aims at integrating applications (considered as services) instead of networks (i.e., protocol stack and/or logical address formats mapping). This service abstraction is an important advantage, since it offers to the application developers an easy and transparent way to integrate the networks with seamless handling of addresses and protocols. Furthermore, the provision of the capability of transparently requesting services in both directions is another benefit offered by our approach. Although sensors are typically providers of sensed data, there are some cases where it is better for the sensor to request a service outside the WSN. That happens, for instance, when the application is too computationally demanding, when it needs to store a huge amount of data, and/or when it requires global knowledge of the WSN.

The rest of this chapter is organized as follows. Section 2 describes a scenario of usage and an overview of the framework. The framework elements, namely Smart, WISeMid, SAGe and Clever, are presented in Sections 3, 4, 5 and 6, respectively. Section 7 discusses an energy consumption evaluation of those elements, and a real application that emphasizes the usefulness of the sensor nodes' capability of being service requestors is also analyzed. Finally, Section 8 presents some concluding remarks.

2. Illustrative scenario

This section presents a scenario of usage of the framework which is depicted in Figure 1 and comprises two distributed applications.

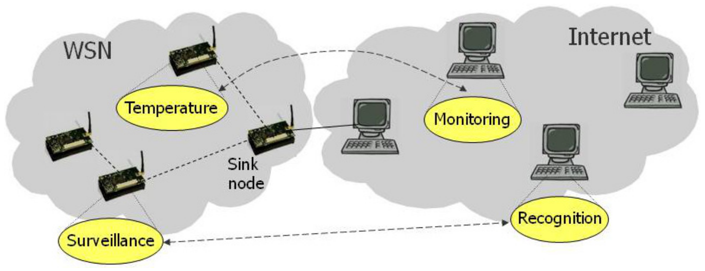


Figure 1. Scenario of usage of the framework

The first application monitors an area (such as a house, a factory, etc.) to detect and report the presence of intruders. It is composed by two services: Surveillance and Recognition. The Surveillance service runs in a sensor node and captures images of any moving target. Each captured image has to be analyzed to verify if the moving target is really a potential threat. This is necessary to avoid reporting the presence of a cat, for example. Therefore, the application has to perform an image recognition procedure to classify the moving target as human or non-human. This kind of task is too computationally demanding and resource consuming, which makes it non-suitable for the resource constraints of the sensor nodes. Considering this, it may be more worthwhile for the Surveillance service to invoke an image recognition service outside the WSN instead of performing itself that task.

Now consider that an Internet host provides the (image) Recognition service, which offers an operation that receives an image, analyzes it and classifies it as human or non-human. The Surveillance service sends the captured image to the Recognition service and, if it signals that the target is a human, the Surveillance service reports the presence of the intruder.

The second distributed application monitors the temperature of an area (such as a forest) to detect high temperatures and prevent potential fires. This application is composed by two services: Monitoring and Temperature (see Figure 1). The Monitoring service runs in an Internet host and keeps track of the environment temperature of the area of interest. For that purpose, it uses a Temperature service that runs in a WSN which has been deployed in that area. This service measures and returns the environment temperature value. Hence, to observe the changes in temperature, the Monitoring service calls the Temperature service periodically.

Both described applications spread out through the Internet and a WSN. In order to allow the construction of this kind of distributed application composed by services from both networks, some issues should be addressed. First, there must be a uniform way of describing those services regardless of their provider being sensor nodes or Internet hosts. Then, based on that description, there must be a mechanism to discover those services and use them. Those services should be able to communicate with each other irrespective of their location. Also, the service location should not influence the way it is accessed. In other words, a service provided by a WSN node should be accessed the same way that a service provided by an Internet host.

Additionally, the distributed applications described above focus on specific areas of interest. The surveillance, for instance, would define the border region of a WSN as a critical one to be monitored, whereas forest fire detecting applications may define clearings that resulted from previous fires as the critical regions to be monitored (since they tend to be more propitious for new fires due to their dry vegetation and soil). Therefore, there should be a mechanism to select a group of nodes in the WSN so such areas can be delimited.

Those issues are handled by the presented framework as described in the following sections.

2.1. Framework overview

For the purpose of enabling the construction of applications that are distributed between WSN's and Internet's nodes, the framework comprises four components: a service model to describe services from both networks; a communication infrastructure (middleware) that enables the interoperability of WSN's and Internet's services; an advanced gateway which is responsible for providing the location and access transparency for the services' communication; and a service composition tool which enables the creation of logical regions in the WSN by just composing Web services that are available on the Internet.

The service model is named Smart (**S**ervice **m**odel for integr**a**ting Wi**r**eless Sensor Networks and the In**t**ernet) [7] and is able to describe both WSN's and Internet's services. Describing a service in a uniform way despite its location is essential to provide the integration of those networks at service level. The Smart model allows a service provider to characterize its service by making available some information about it. That information includes functional details, such as the service interface, nonfunctional details, such as service properties, and information on how to interact with it so the service can be accessed.

Once the services have been described, a middleware called WISeMid (**W**ireless sensor network's and **I**nternet's **S**ervices int**e**gration **Mid**dleware) [8] provides a communication infrastructure for the services interaction, supporting the integration of WSNs and the Internet at service level. In this context, applications running in the Internet/WSN nodes may play the role of service providers or service users, where a service user is able to communicate with a service provider no matter whether they are running in the same network or not. For that purpose, WISeMid provides an infrastructure that allows integrating these services in such a transparent manner that services provided by both WSN nodes or Internet hosts are accessed the same way. Moreover, WISeMid implements the following mechanisms for saving energy in WSNs [9]: Aggregation service, which aggregates the last `n` data sensed by a node; Reply Storage Timeout, which avoids sending equivalent messages to the sensor nodes while the last sensed data is still considered up-to-date; Automatic Type Conversion, which removes unnecessary bytes from the messages; and the implementation of asynchronous invocation patterns, which prevents the sensor application from wasting power for being blocked during a service request.

The access and location transparency provided by the middleware is performed by the gateway, which is called SAGe (**S**ensor **A**dvanced **G**at**e**way) [10]. Running in an Internet host which is connected to the WSN sink node, SAGe's main function is to act as a service proxy between both networks by enabling the communication between services running on Internet hosts and WSN nodes in such way that the service user should invoke a service without knowing if it is being provided by a WSN node or an Internet host. Additionally, SAGe is involved in all energy-saving methods performed by WISeMid.

The service-level integration of the WSNs and the Internet is enhanced by a Web service composition tool named Clever (**C**omposing **l**ogical r**e**gions **v**ia s**er**vices). This tool enables the definition of logical regions in WSN by composing Web services available in the Internet. Having a Web service acting as a proxy to each WSN node, the definition of a logical region in the WSN consists of defining a Web service composition. Once that composition is deployed, a new Web service is created and an invocation to it is passed to all sensors belonging to the WSN logical region and represented by the composition.

The next sections will describe in details each of those components.

3. Smart

The first element of the framework is Smart (**S**ervice **m**odel for integr**a**ting Wi**r**eless Sensor Networks and the In**t**ernet), a service model that is suitable for both Internet and WSN services, promoting the integration of those networks. In other words, both services designed for a WSN and services designed for the Internet may be described using the Smart model. Although there are many service models for the Internet [11, 12] and a few service-oriented models for WSNs [13, 14], for the best of our knowledge, Smart is the first service model able to describe both WSN and Internet services.

Either on the Internet or in a WSN, a service provider needs to characterize its service by making available some information about it. The essential and simplest information that must be available is the service identification and what it can do. The service interface offers that information by specifying the service name and the operations it provides. A description of how the service provides its operations may also be exposed. That information concerns the service functionality and thus compose the service functional description.

A service user may be interested in knowing specific details of the service provider, the service implementation or its running environment. That information is called nonfunctional description and may include, for instance, the bit rate of a service provider or its location.

Furthermore, for a service to be accessed, information on how to interact with it needs to be defined. That is so called interoperability information and includes the format of the messages the service understands and its access point.

With that in mind, the Smart model defines a service as being composed by functional and nonfunctional descriptions, from which depends its interoperability information.

3.1. Functional description

To be able to request a service and the execution of one of its operations, a service user (client) needs first to be aware of the service name and what operations it can perform. More than that, the client needs to know the input and output data of a successfully execution of the service, as well as the possible errors that an unsuccessful invocation may generate in order to handle them. That information is commonly referred to as the service interface.

Some services share the name and the operations they provide, but may produce different results by performing different actions or being under different conditions to execute the same operation. Those services are said to have the same type, and the differences between them are expressed using properties, which assume different values from service to service.

Another service characteristic that concerns its functionality is the period of time the service instance needs to exist, which is known as its lifecycle and is closely related to maintaining the service state between subsequent invocations. According to the lifecycle patterns defined in [15], service instances may be static, per-request or per-client. A static instance may be used when the service is independent of any service user (client), and its state must be available to all clients between individual invocations. When the service does not require maintaining state and is accessed by many users at the same time, an instance per request is more suitable. A per client instance is appropriate when it is important to maintain the service state for subsequent invocations by the same user, which happens when the logic of the service extends the logic of the client.

Considering all that information about the service functionality, the Smart model defines the service `FunctionalDescription` as being composed by its `Interface` and its `Behavior`, as shown by the class diagram in Figure 2. Those elements are explained in detail as follows.

3.1.1. Interface

As previously stated, the interface contains the service name and the operation(s) it provides (see Figure 2). The service's `name` is used by a service client to locate a particular service.

Concerning the service operations, each `Operation` is defined by a `name`, an operation `type`, a list of typed parameters (`Argument`), a result (`Outcome`) and a list of possible errors (`Fault`). There are two types of operation: `Interrogation`, which is a request-response operation; and `Announcement`, which is an one-way operation. The list of errors represents exceptions raised by the operation, each of which is defined by a `name` and a `message`.

In the WSN context, an operation is either a `Command` or an `Event`, which is actually a feature of nesC, one of the most used programming language for developing applications for WSNs [16]. Commands are implemented by the interface's provider, whereas events are implemented by the interface's user.

The `ServiceType` is used to categorize services according to their capabilities. That means that all services of the same type, share the same `interface` and the same `properties`,

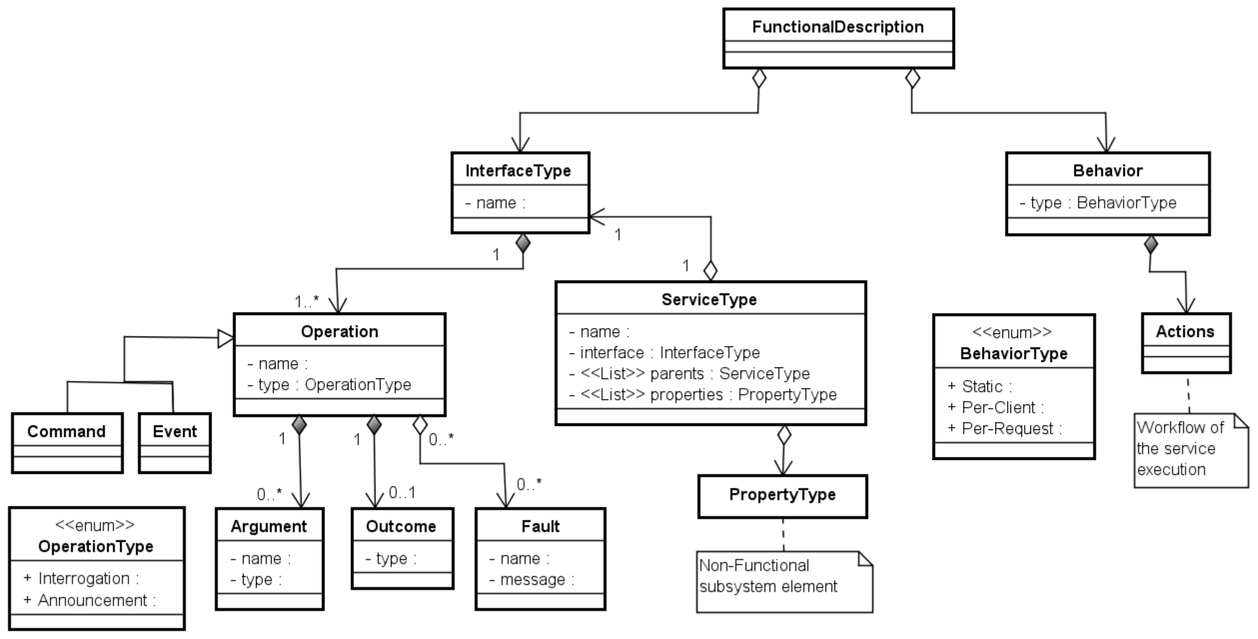


Figure 2. UML Class diagram of the Functional description

being distinguished by the different values of their properties. Also, a service type may extend another one (`parent`), inheriting its interface signature, while being able to add more operations, and its properties, also being able to define additional ones.

3.1.2. Behavior

The service behavior includes a `BehaviorType` and a set of `Actions`, as depicted in Figure 2. The behavior type defines the service's lifecycle: static, which specifies that the service has a unique instance; per-client, which states the service has an instance for each client; and per-request, meaning that each request is handled by a different instance of the service.

The `Actions` element defines the service behavior in terms of activity structure. It specifies the workflow (execution) of the service. This workflow may be simple, specifying the expected execution order of the service operations, which is necessary when the execution of an operation affects the execution of other one. It may also be more complex, identifying the actions of the operation and specifying the ordered sequence in which they are performed.

3.2. Nonfunctional description

The functional description specifies *what* the service can do, but describing *how* the service can do it may be very useful or even necessary for a service user to choose the service that best fits their needs. This extra description includes characteristics (properties) that are not directly related to the functional description of the service, such as the server geographic location or the service security, and therefore is commonly referred to as nonfunctional description.

A particular subset of nonfunctional characteristics is very important for choosing the proper service: quality of service (QoS) parameters. Although the term QoS is typically related to network performance parameters (such as bandwidth, latency and error rate), in a service related context it covers a wider range of service properties which can be used as quality indicators, including, for instance, accuracy, dependability, robustness, security, customer service, etc.

With that in mind, Smart defines that the `NonfunctionalDescription` of a service specifies its nonfunctional properties (`Property`) and QoS parameters (`QoSParameter`), as depicted in Figure 3. Those elements are explained in detail next.

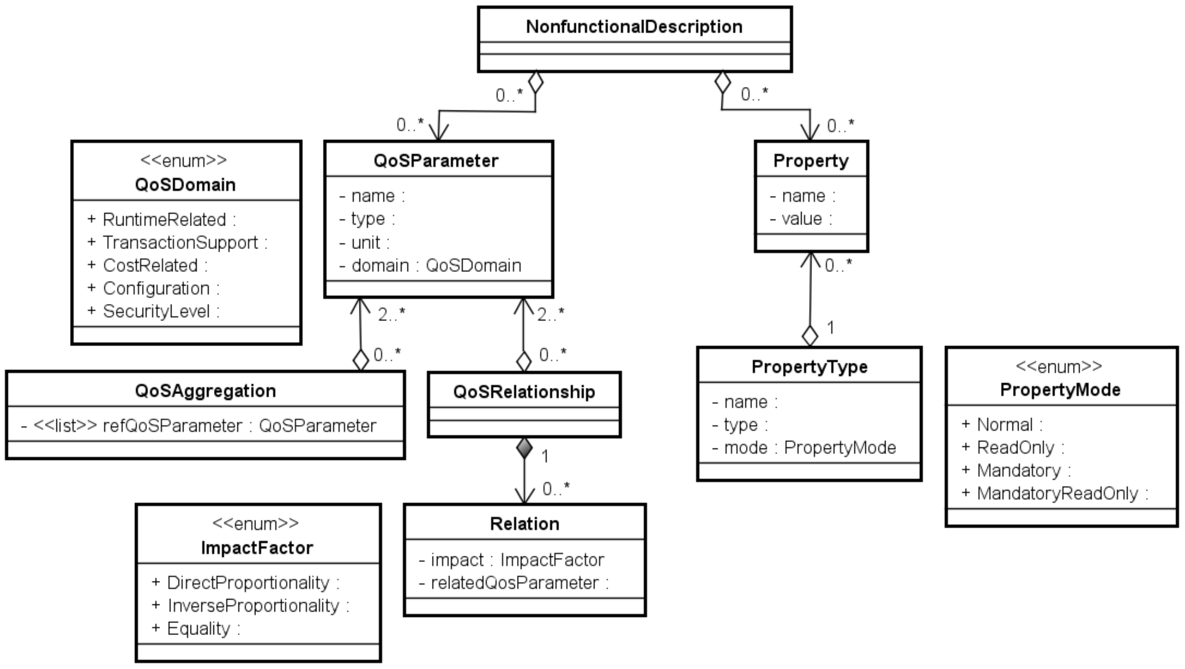


Figure 3. UML Class diagram of Nonfunctional description

3.2.1. Property

The Smart model elements which describe properties (`Property`, `PropertyType` and `PropertyMode`), are based on the CORBA Trading Object Service Specification [17].

In Smart, each `Property` can be described by a name-value tuple, where `name` is a string that names the property and `value` is the value assigned to the property (see Figure 3).

The service properties are actually classified in property types, which are related to the `ServiceType` element of the functional description (see Figure 2). A `PropertyType` is defined by a `name`, a property value `type` (e.g., integer, string, etc.) and a `mode`. The mode (`PropertyMode`) defines whether a property is mandatory and/or readonly. When a property is `Mandatory`, it means that an instance of the service type that defines this property must provide an appropriate value for it when registering the service. When it is `ReadOnly`, the value for this property is optional, but if provided, it may not be changed after the service is registered. When it is both mandatory and readonly (`MandatoryReadOnly`), the property value should be provided when registering the service and may not be changed after that. Finally, when a property is `Normal`, it is optional and its value may be subsequently modified.

3.2.2. QoS Parameters

QoS parameters refer to service properties that can be used as indicators of quality, as said previously. Based on the SMEPP service model description [18], the Smart model defines that a `QoSParameter` is described by a `name`, a `type`, a `unit` and a `QoSDomain` (see Figure 3). The `name` is a string that specifies the QoS parameter name, such as "throughput". The `type` describes the data type of the QoS parameter value (e.g.: integer, float, etc.), whereas the `unit` represents the unit of measurement (such as second, bits, percent, etc.).

A `QoSDomain` specifies the domain of the information enclosed by the QoS parameter. Possible QoS domains include runtime (e.g.: performance), transaction support (e.g.: integrity), configuration and cost (e.g.: stability), and security (e.g.: authentication) [19].

Some QoS parameters are composed by two or more parameters. For example, the "performance" parameter is composed by "response time", "throughput" and "latency". Such compound QoS parameters may be described by a `QoSAggregation`, with two or more QoS parameters, each of which represents a parameter member of the composition.

Also, QoS parameters may affect other ones and this influence is expressed by the `QoSRelationship`, which defines the impact factor as one of the following values: `DirectProportionality` (i.e., the related parameter values have the same behavior: if one increases or decreases, so does the other), `InverseProportionality` (i.e., the parameter values have the opposite behavior: if one increases, the other decreases; and vice-versa) or `Equality` (i.e., the parameters have the same value).

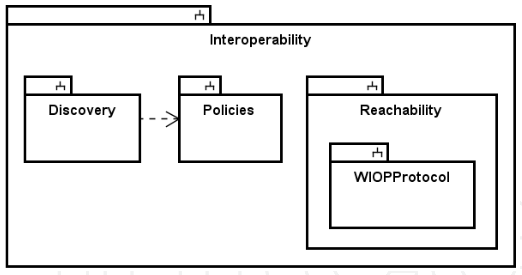
3.3. Interoperability information

According to [20], interoperability *"is about the meaningful sharing of functionality and information that leads to the achievement of a common goal"*. For that to be possible, a message exchange pattern should be defined by the model. It specifies the message sequence and the exchange rules, which should be followed by both sides of the interaction and is usually specified by a protocol definition. Also, for the interoperability to happen, the services must provide a kind of identification, a way to be accessed.

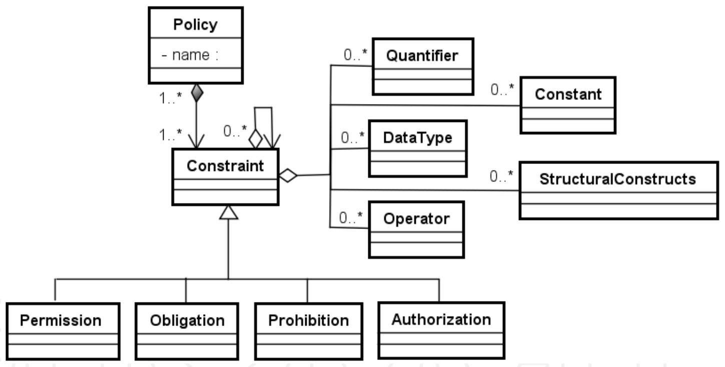
Before being able to exchange messages, the services need to find each other. By finding, we mean searching and retrieving information about the services which satisfy some user defined conditions. The service discovery procedure is responsible for that task and is implemented by some special services that offer operations for registering and searching services. The most simple discovery service is called Naming service and locates a service based only on the service name. Trading service is another special service that implements a discovery procedure, which discovers a service based on its type and a set of property values that describe it.

Still concerning service interoperability, a set of policies may be defined to guide the services interaction. Different policies may be involved in the process of service discovery to check the constraints that either the service provider or the service user specify to protect the process of communication or sharing the information.

In Smart, all that information that concerns the way a service may interact with another is represented by `Interoperability` element, which consists of `Policies`, `Discovery` and `Reachability`, as illustrated in Figure 4(a).



(a) UML Subsystem diagram of Interoperability information



(b) UML Class diagram of Policy

Figure 4. UML diagrams of Interoperability information

3.3.1. Policy

As stated previously, policies may be used by either the service provider or the service user to protect the process of communication or sharing the information. For that end, each `Policy` defined by a service prescribes a set of `Constraints` for interacting with it, as depicted in Figure 4(b). Those constraints define the ideal, acceptable or desirable service behavior during an interaction with another service.

In the Smart model, a `Constraint` may describe an `Obligation` – a behavior that is required; a `Permission` – a behavior that is allowed to occur; a `Prohibition` – a behavior that must not occur; and an `Authorization` – a behavior that must not be prevented for the services involved. Note that a permission is equivalent to there being no obligation for the behavior not to occur, whilst an authorization is actually an empowerment.

Constraints may contain quantifiers (e.g.: for all, there exists), constants, data types (e.g.: integer, String), operators (arithmetic operators, comparison operators, boolean operators, implication operators) and structural constructs (e.g.: let in, if then else) [21].

3.3.2. ReachabilityInfo

The `ReachabilityInfo` element is in charge of providing the service information that enables other services to locate and interact with it. Therefore, it should define the `EndPoint` to which a client can direct messages to invoke actions and the `MessageExchangePattern`, which specifies the protocol to adopt for message exchange using the endpoint (see Figure 5).

Since Smart aims at integrating the Internet and WSNs, it specifies two kinds of end point to a service, one for each network. For an Internet service, the `InternetEndPoint` is described by the IP address of the host (`ipAddress`) and the port number through which the service is provided (`portNumber`). For a WSN service, the `WsnEndPoint` may identify a `nodeId` or a `groupId`, as a service may be provided by a single node or by a group of nodes.

For the `MessageExchangePattern`, the Smart model defines that the `Publish/Subscribe` communication model or the `Request/Reply` may be used. The former has been chosen due to its suitability in WSNs, since it saves energy by sending a message only when an event of interest is detected. The request/reply approach has been chosen for being the most suitable for server/client communication. For now, only the

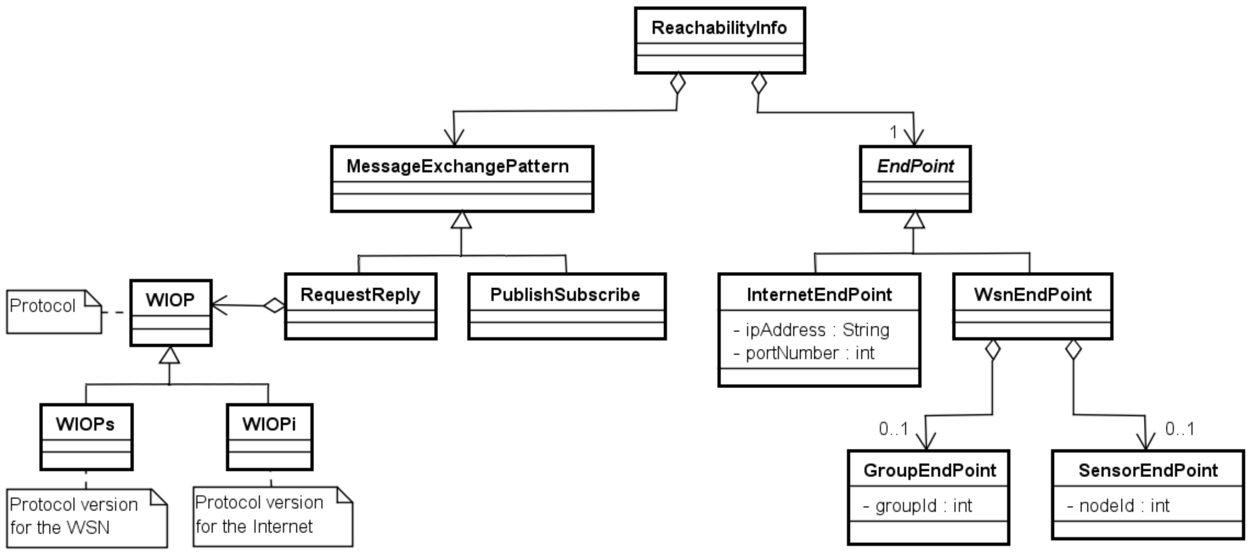


Figure 5. UML Class diagram of Reachability information

request/reply has been detailed in the model, with a protocol being defined for that purpose. The request/reply protocol, namely `WIOP` (see Section 4.3.1 for details), defines different message formats for Internet service (`WIOPi`) and WSN services (`WIOPs`), to handle their specificities, and provides the mapping among those formats (especially concerning the data types) in order to ensure that the exchanged information has the same meaning for both message sender and receiver, no matter if one is located at the Internet and another at a WSN.

3.3.3. DiscoveryProcedure

As its name suggests, the `DiscoveryProcedure` element models the service discovery procedure, which may be defined as the act of finding a service that may have been previously unknown and that meets certain functional criteria. Since additional nonfunctional criteria may be used to locate the desired service, the `DiscoveryProcedure` element interacts with the FunctionalDescription and the NonfunctionalDescription elements, as depicted in Figure 6.

Smart defines two discovery procedures: the `Naming`, that uses only functional criteria, and `Trader`, that uses functional and nonfunctional criteria. Both procedures are detailed next.

3.3.3.1. Naming

The `Naming` component represents the Naming discovery procedure, which is used to register/locate a service based on its name. To provide scalability to the Naming Service, the Smart model defines two architectures for it: `Flat` and `Hierarchical` (see Figure 6). The flat Naming uses unique, globally distinguished names, whilst the hierarchical one deals with different domains. Therefore, the chosen architecture defines the name space, that comprises a `Context` in which names are unique and valid.

3.3.3.2. Trader

The `Trader` component represents the Trading discovery procedure. The Smart model definition of the trading function is based on the CORBA Trading Object Service Specification [17] and the ANSA Model for Trading and Federation [22].

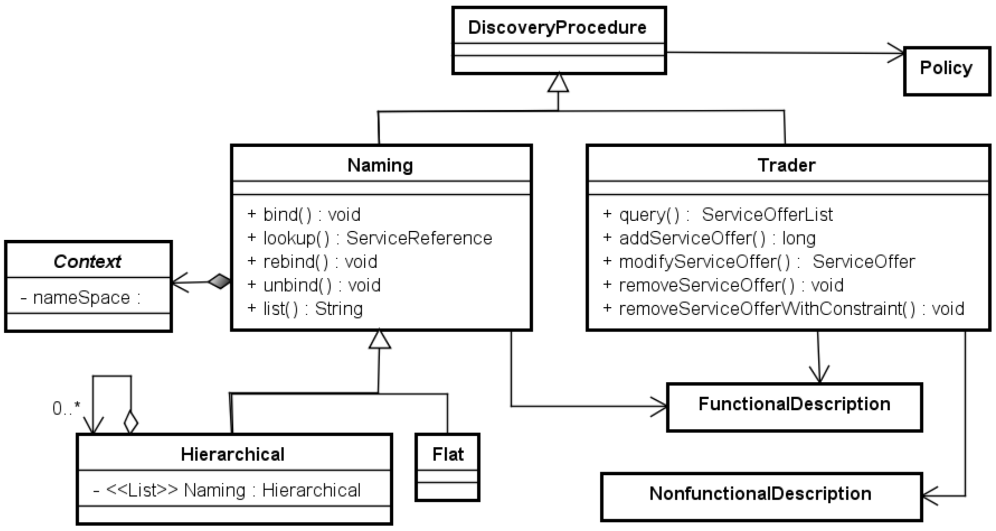

DiscoveryProcedure
Policy
Naming
+ bind() : void
+ lookup() : ServiceReference
+ rebind() : void
+ unbind() : void
+ list() : String
Trader
+ query() : ServiceOfferList
+ addServiceOffer() : long
+ modifyServiceOffer() : ServiceOffer
+ removeServiceOffer() : void
+ removeServiceOfferWithConstraint() : void
Context
- nameSpace :
0..*
Hierarchical
- <<List>> Naming : Hierarchical
Flat
FunctionalDescription
NonfunctionalDescription

Figure 6. UML Class diagram of Discovery

As stated before, the trading service is concerned with discovering a service based on its type and a set of property values that describe it. The act of registering information about a service in the Trader is usually called exporting a service offer. A service offer consists of the service type name, a reference to the interface that is provided by the service, and zero or more property values for the service. Those properties correspond to the ones that are listed in the associated service type (and parents), and specifying a value for them depends on the mode assigned to each one of them. A value must be provided for every property that has been defined as mandatory (including both Mandatory and MandatoryReadOnly modes) in the service type. Providing a value for the other properties is optional.

4. WISeMid

As mentioned earlier, WISeMid provides a communication infrastructure for services interaction, where applications running in the Internet/WSN nodes act as service providers or service users that are able to communicate irrespective of being at the same network or not. As depicted in Figure 7(a), those applications (services) should be developed in different technologies, such as Web Service, Java RMI, EJB and JMS, although WISeMid current implementation supports only WISeMid services and Web services.

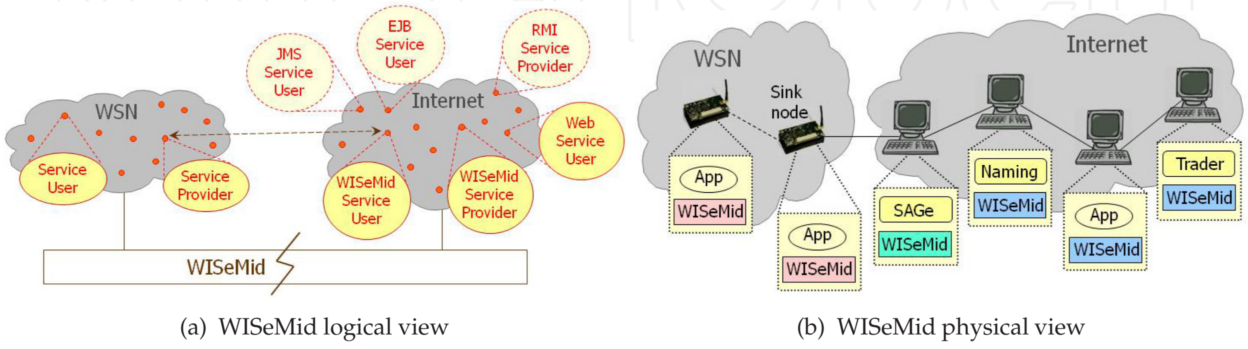


Figure 7. WISeMid views

Figure 7(b) presents a physical view of how WISeMid spreads out through the Internet and WSN. The WISeMid implementation for WSN and Internet is not the same, as they have different requirements and components (that will be explained in the subsequent sections). Physical communication is performed through an Internet host that is connected to the WSN sink node via a serial port (USB). This host executes a special WISeMid service called SAGe, which acts as a proxy between both networks (see Section 5).

4.1. WIDL

The first step to enable the services interaction is to specify a notation to describe a service. For this particular purpose, we have defined the WISeMid IDL – WISeMid Interface Definition Language. It enables us to define service interfaces in a uniform way, wherever the service runs (Internet or WSN) and whatever the implementation language (Java or nesC). Based on Smart model definition of the service interface and its properties (see Sections 3.1.1 and 3.2.1), the general structure of a service definition in WISeMid IDL is presented below:

```
1: module PACKAGE_NAME{
2:   interface INTERFACE_NAME{
3:     [OPER_TYPE] OUTCOME_TYPE  OPER_NAME([TYPE ARG1,...]) [raises(EXCEPTION_NAME1,...)]
4:   }
5:   [type TYPE_NAME [extends TYPE_NAME] {
6:     propertyType [PROP_MODE ,] PROP_NAME , PROP_TYPE;
7:   }]
8:   [properties TYPE_NAME {
9:     [property PROP_NAME = VALUE;]
10:  }]
11:}
```

First, the module (package) that contains the service should be specified (1), followed by the service interface, which includes its name (2) and provided operations (3). Each operation has a name, input/output typed parameters and may raise exceptions. An operation is by default a request-response operation, but it may be defined as a one-way operation by using the reserved word `oneway` as the operation type. (Note that in Smart, request-response corresponds to the `interrogation` type and one-way is the `announcement` type.) The service type definition (5-7) includes its name, its parent's name (5), and a list of property types, each of which comprising a property mode (optional), a name and a type (6). The property mode options are `NORMAL` (default), `MANDATORY`, `READ_ONLY` and `MANDATORY_READ_ONLY`.

Property values are defined in `properties` clauses (8-10), which are related to the service type that specified the property type (8). Every property type defined as `MANDATORY` or `MANDATORY_READ_ONLY` in the service type must have a property value associated to its name in the `properties` clause (9). Specifying a value for property types of other modes is optional. Additionally, property values that have not been defined in the service type may be specified. The definition of service type and properties is optional.

The service definition in a WIDL file may be used for automatic code generation. To perform that task, we have developed a tool, namely ProxiesGen, that can interpret a WIDL file and generate the proxies (i.e., stub and skeleton) related to the described service according to a command line parameter, which specifies if the service is supposed to be provided by a WSN node or by an Internet host. When service type and properties are defined, ProxiesGen adds to the server code the creation and registration of the service type in the Trader. Also, a service offer with the defined property values is created and exported.

4.2. Architecture

Based on the DOC middleware layers [23], the WISeMid architecture consists of three layers: Infrastructure, Distribution and Common Services (see Figure 8).

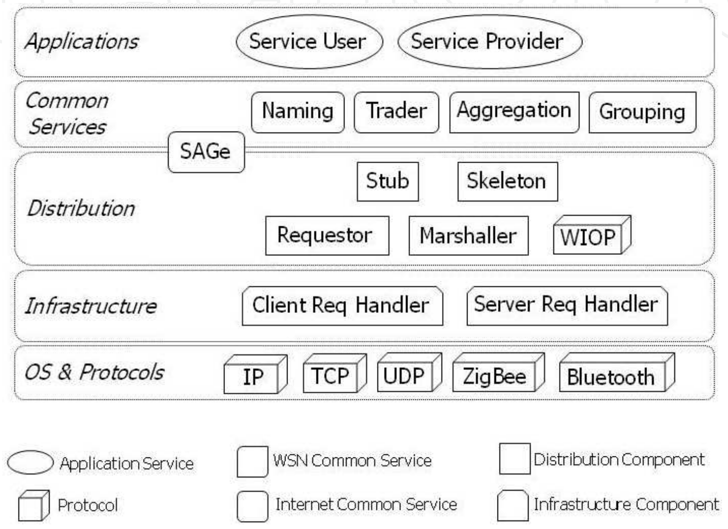


Figure 8. WISeMid Architecture

The Common Services layer includes services that are not particular to a specific application domain: Aggregation, which performs sensor data aggregation; Grouping, which defines clusters inside the WSN; Naming and Trader, that store information needed to find and access a service (they run in the Internet); and SAGe, that is in charge of converting and forwarding messages from/to WSN (it runs in the Internet). Additionally, SAGe also provides location transparency acting as a service proxy between both networks, as described in Section 5.

The Distribution layer includes the following elements: Stub, Requestor, Skeleton and Marshaller, which are basic middleware elements [15]. Those elements handle WIOP messages. WIOP is the request/reply protocol specified by Smart (see Section 3.3.2).

The Infrastructure layer consists of the Client Request Handler and the Server Request Handler, which handle network communication using the communication facilities provided by the operating systems, e.g., sockets (Windows) and ActiveMessageC (TinyOS).

4.3. Implementation

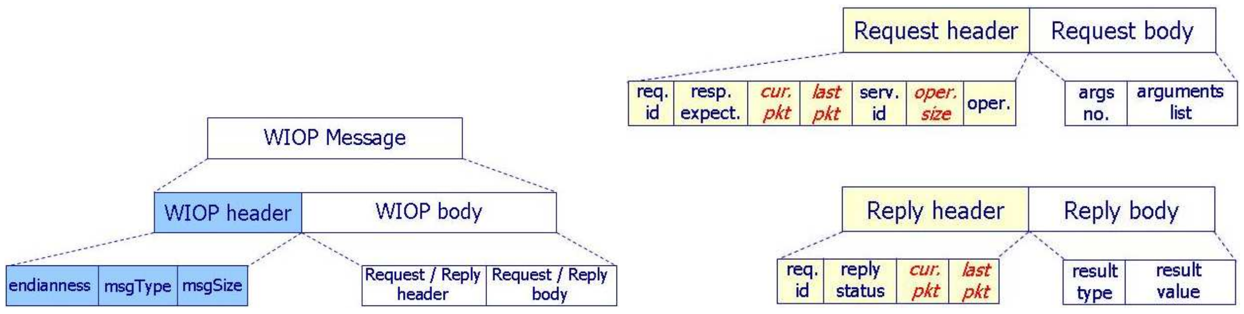
The WISeMid implementation is divided into two parts, one for the WSN nodes, developed in nesC, and another for Internet hosts, developed in Java. The elements of the infrastructure layer (Client Request Handler and Server Request Handler) and of the distribution layer (Stub, Skeleton, Requestor, Marshaller and WIOP) have been implemented in both languages, as nodes from both networks may play the role of user or provider of the service. Actually, due to sensor nodes' limited resources, some elements are not present in the WSN: the Requestor is not implemented by the WSN service users (its functions are deployed by the Stub), and WIOP messages are treated as byte sequences, making the Marshaller unnecessary.

The above elements, except for WIOP, are basic middleware elements, thus their implementation will not be described here. The Aggregation and the Grouping elements are common services that run in the WSN. The Aggregation service may be considered as a method to save energy in the sensor nodes and therefore it is described in Section 4.3.4 with other energy-saving approaches that have been implemented in WISeMid. The Grouping service was implemented as a web service composition approach that enables grouping the WSN nodes into logical regions. This approach is used by Clever and described in Section 6.1.

As SAGe has an essential role in WISeMid, it is described in Section 5. The other architecture elements (WIOP, Naming and Trader) are described as follows.

4.3.1. WIOP

Specified by the Smart model as the message exchange pattern to enable service interoperability (see Section 3.3.2), the WISeMid Inter-ORB Protocol (WIOP) is a GIOP-based protocol that defines the Request/Reply messages between clients and servers. A WIOP message is divided into header and body, as depicted in Figure 9(a).



(a) WIOP Message header (b) WIOP Message body

Figure 9. WIOP (WISeMid Inter-ORB Protocol)

The WIOP message header is composed by the following fields: `endianness` (e.g., big endian or little endian); `msgType` (e.g., request or reply message); and `msgSize`, which stores the message size in bytes. It is worth noting that WIOP messages do not contain fields for source/destination ID (such as sensor ID or IP address). That happens because, as other inter-orb protocols (e.g. GIOP), WIOP uses the protocols of the lower layers in both networks (that is, Active Message in WSN and TCP/IP in the Internet), and WISeMid infrastructure, more specifically SAGe, handles the information concerning addressing.

The WIOP message body may contain a Request or a Reply message. Those messages, which have also a header and a body, are illustrated in Figure 9(b).

The Request message header's main fields are: `requestId`, which stores the Request message ID; `responseExpected`, which signals whether the request expects a Reply message or not; `serviceId`, which is the ID of the requested service; and `operation`, which represents the name of the operation being invoked. Those are the fields that compose the Request message header in the Internet version of WIOP (called WIOP$_\mathrm{I}$). The version that runs in the WSN (called WIOP$_\mathrm{S}$) contains three additional fields (emphasized in Figure 9(b)):

currentPacket and lastPacket, which are the current and last packet ID, respectively, and operationSize, which represents the length of the operation name. This last field is necessary because the operation field has no fixed size. It is adjusted to the length of the operation name, to keep the message as minimal as possible. Furthermore, a single WIOP$_s$ may not be enough to transmit the total amount of data to/from a WSN service, due to the size limitation imposed by TinyOS' Active Message [24]. In such cases, the data is divided in many WIOP$_s$ messages, and the currentPacket and lastPacket fields are used to control the message fragmentation/reassembly at the sensor node and SAGe.

The Request body consists of the number of arguments (numArgs) followed by a sequence of type and value of each argument. Also, the way arguments are stored in the Request body is different for the WSN version. In the WIOP$_i$, the arguments are stored one by one, being each argument composed by a type and a value. For example, three arguments would be stored in the following sequence: type1, value1, type2, value2, type3, value3. In the WIOP$_s$, only the first argument is individually stored (with its type and value in a row). From the second argument on, the arguments are grouped into pairs where the types of both arguments come first followed by their respective values. That happens because types are represented by integer numbers between 0 and 11, and therefore each type can be stored in only 4 bits. Hence two types can be grouped into one byte, being followed by their related argument values. The first type is a special case as it uses the second half of the byte that carries the numArgs field. Considering this configuration, three arguments would be stored in the following sequence: type1, value1, type2, type3, value2, value3.

The Reply message header main fields are: requestId, which stores the related Request message ID; and replyStatus, which signals whether there was any exception while executing the request, and its possible values are NO_EXCEPTION (0), USER_EXCEPTION (1), SYSTEM_EXCEPTION (2) and LOCATION_FORWARD (3). As the Request message header, the Reply message header in the WIOP$_s$ contains the additional fields related to the message fragmentation/reassembly: currentPacket and lastPacket. The Reply body is composed by the result type and its value for both formats: Internet and WSN (see Figure 9(b)).

Besides saving energy by its reduced size, the sensor message format also concerns about sensor limited processing as it is already deployed as a byte array, avoiding the need for a Marshaller implementation.

The WISeMid Naming Service, the Trader and Internet services use the Internet format (WIOP$_I$), while the sensor services use the WSN format (WIOP$_S$). Only SAGe handles both formats.

4.3.2. Naming service

The WISeMid Naming Service implements the Naming discovery procedure defined by Smart in Section 3.3.3.1. It stores the references of services executing in the Internet and WSN in such a way that a service may only be accessed/used after being registered in the Naming Service.

The Naming Service's interface includes five operations: Bind, to register a service by its name, associating it with its reference; Lookup, to return the reference associated to a service name; Rebind, to change the reference that is associated with a service name; Unbind, to unregister a service name; and List, to list all registered services. The service reference includes the service

ID, endianness and the Internet end point of the service, which consists of its IP address and port number. Note that, since the Naming service runs in the Internet, the service references handled by it have no information about WSN end points. Nevertheless, WSN services may also be registered using the Internet end point of SAGe, which handles another type of service reference to store WSN end point information, as described next. It is worth mentioning that WISeMid Naming Service has a flat architecture, with a global name space.

4.3.3. Trading service

WISeMid's Trader implements the Trading service, another discovery procedure described by Smart (see Section 3.3.3.2). It has a graphical user interface (GUI) that allows users to define service types and property types, as well as export and query service offers.

The service type comprises a name, a reference for the service interface, a parent and a list of property types (those last two are optional attributes). When defined, each property type comprises a name, a property value type (e.g., integer, string, float, etc.) and a mode (which defines whether a property is mandatory and/or readonly) (see Section 3.2.1).

A service offer is the information that is registered in the Trader about a service and the act of registering that information in the Trader is called exporting a service offer. As stated in Section 3.3.3.2, a service offer consists of the service type name, a reference to the interface that provides the service, and zero or more property values for the service, where those properties correspond to the ones that are listed in the associated service type and specifying a value to them depends on the mode associated to each one of them. A value must be provided for every mandatory property (both `Mandatory` and `MandatoryReadOnly` modes) of the service type. On the other hand, providing a value to the other properties is optional, and properties that have not been specified in the related service type may be added in the service offer (with values being provided to them). WISeMid implements all those characteristics and adds a service offer identification number to uniquely identify a service offer.

4.3.4. Energy-saving methods

Since energy is a scarce resource in WSNs, performing energy-saving methods is essential to extend the sensor nodes lifetime and thus the WSN lifetime. This section presents some energy-saving approaches that have been implemented in WISeMid [9].

4.3.4.1. Aggregation

Besides this in-network data aggregation, which is used by most middleware, WISeMid implements an aggregation service which aggregates the last `n` data sensed by a node. The effect of this service is to aggregate results of services provided by a sensor node, sending only one (1) reply message instead of `n`. In addition to avoiding the transmission of `n-1` reply messages, this procedure also eliminates the need of sending `n-1` requests. The service user may send just one request to receive the same `n` values but in an aggregated form.

4.3.4.2. Reply storage timeout

Considering that some physical aspects sensed by a sensor node, such as temperature, do not present a great variability in terms of second/minute time scale, this approach avoids sending equivalent Request messages (that is, messages asking for the same service with the same parameters) during a short period of time, as the returned data are likely to be the same. Hence, SAGe groups equivalent Request messages and, for a configurable period of time, only one Request is sent to the sensor service provider, and the received Reply message is stored and forwarded as an answer to all the equivalent Request messages that arrive during that period. For the cases where the sensed value changes very often, this procedure may be turned off by setting to null (i.e., 0 seconds) the Reply message storage timeout.

4.3.4.3. Automatic type conversion

Since the transmission/reception of data is very energy consuming, the more data is sent/received by a sensor, the more energy is spent. With that in mind, SAGe performs an additional step when converting an Internet Request message into a sensor Request message. For each argument in the Request body, it tries to fit the argument value in a smaller type (that is, a compatible type that uses less bytes). For instance, if the argument is a `long` (an integer of 8 bytes) but its value is '123', it can be stored into a `byte` (an integer of 1 byte). Thus SAGe converts the argument from a `long` into a `byte` and adds only 1 byte to the WIOP$_s$ instead of the original 8 bytes, avoiding the transmission of 7 bytes. The same step is performed for the result value of WIOP$_i$ Reply messages when converting them into WIOP$_s$ Reply messages.

4.3.4.4. Asynchronous invocation patterns

Initially, the WISeMid services communicated synchronously. However, synchronous invocation blocks the service user (client) until the result returned from the service provider (server) is received. Thus, when a service user is running in a sensor node, it keeps consuming energy while waiting for the answer without executing any task. For that reason, we extended WISeMid with asynchronous invocation patterns, which enable the client to resume its work immediately after a remote invocation is sent. Four asynchronous invocation patterns, which are presented in [15], have been implemented in WISeMid, namely: Fire and Forget, Sync with Server, Poll Object and Result Callback. The first two patterns are used for one-way operations, while the last ones are used for request-response operations. Details on WISeMid implementation of those patterns can be found in [9].

5. SAGe

As stated previously, SAGe is an important WISeMid service which performs different tasks to allow the integration of the Internet and WSNs.

SAGe may be considered an advanced version of SerialForwarder [24], a TinyOS application which allows multiple clients to communicate with a mote working as a base station. SAGe extends SerialForwarder functionalities by performing some data processing before forwarding the packets from the WSN (more specifically, the sink node) to the connected clients and vice-versa. Running in the Internet host connected to the WSN sink node via a serial port (see Figure 7(b)), SAGe's main function is to act as a service proxy between both networks by enabling the communication between services running on Internet hosts and WSN nodes in a transparent way. Furthermore, SAGe also performs some tasks concerning the limited resources of sensor nodes, such as reducing the messages size and avoiding sending unnecessary messages to the WSN. The rest of this section describes SAGe architecture and functions.

5.1. SAGe architecture

The SAGe architecture, which is illustrated in Figure 10, comprises three components:

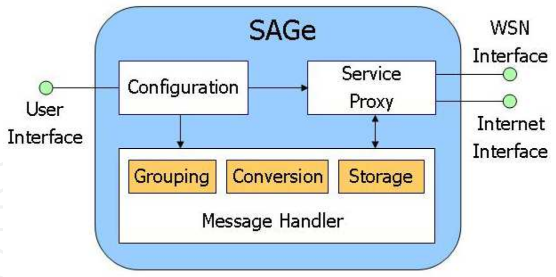


Figure 10. SAGe architecture

- **Configuration:** provides a user interface (UI) that enables to configure some SAGe attributes, such as: the port number used by Internet clients to connect to SAGe, and the period of time that a sensor Reply message should be considered an up-to-date information (to refrain SAGe from sending unnecessary Request messages to WSN).
- **Service Proxy:** provides one interface per network (WSN and Internet) through which SAGe receives/sends WIOP messages to provide communication between the services.
- **Message Handling:** handles WIOP messages and is divided into three subcomponents:
  - **Grouping:** groups equivalent Internet Request messages during a given period of time (set by Configuration) to avoid WSN energy waste;
  - **Conversion:** performs the conversion between Internet and WSN WIOP message formats (namely, $WIOP_i$ and $WIOP_S$);
  - **Storage:** stores the messages received from both networks during a given period of time (set by Configuration), in order to enable message grouping.

Those (sub)components cooperate to perform SAGe functions, which are described next.

5.2. SAGe functions

As mentioned above, the main functions of SAGe are acting as a service proxy and participating in the energy-saving mechanisms.

In order to enable the interaction of services of both Internet and WSN in a transparent way while acting as a service proxy, SAGe has an important role in three specific tasks: the binding of a WSN service in the Naming Service, the invocation of a WSN service (provider) by an Internet service (user), and the invocation of an Internet service (provider) by a WSN service (user). Those three tasks as well as SAGe's participation the methods to conserve the WSN nodes energy are described as follows.

5.2.1. Binding of a WSN service

Once a WSN service starts, it sends a message invoking the Bind operation of the Naming Service. When SAGe receives that message, it creates a `ServiceReference` to the WSN service including the SAGe's IP address and port, and then sends a bind request to the Naming Service, registering the WSN service as a SAGe service. It also keeps the created reference cached as a `SageServiceReference`, which assigns the `ServiceReference` to the node ID of the sensor providing the service. In such manner, SAGe knows which sensor node a Request message to a WSN service must be forwarded to.

5.2.2. Invocation of a WSN service

When the SAGe receives a Request message from the Internet invoking a WSN service, it converts the message to a WIOP_s Request and sends it to the WSN using the `SageServiceReference` that has been cached when the WSN service was bound. Once the Reply message from the sensor service provider is received, SAGe converts it into a WIOP_i Reply message and forwards it to the Internet service user. If no `SageServiceReference` is found, SAGe does not forward the request to the WSN since no sensor provides this service. Instead, it sends a Reply message reporting an error to the Internet host that requested the service.

5.2.3. Invocation of an Internet service

When a sensor node service performs a lookup for an Internet service, SAGe checks if this service is already known, i.e., if its reference is cached. If the service is unknown, SAGe converts and forwards the lookup request to the WISeMid Naming Service. When it receives the WIOP_i Reply, it stores the returned `ServiceReference` and sends the service ID to the sensor node service. Using the received service ID, the WSN service invokes the Internet service operation. When SAGe receives the sensor Request message, it uses the cached `ServiceReference` to invoke the requested operation and, once the Reply message arrives, SAGe converts and forwards it to the sensor node service user.

5.2.4. Saving WSN limited resources

As exposed in Section 4.3.4, WISeMid implements some energy-saving mechanisms. SAGe has an essential role in some of these mechanisms, as explained below.

In the Reply Storage Timeout mechanism, SAGe is in charge of everything: allowing the configuration of the period of time the Reply message will be considered up-to-date; recognizing and grouping equivalent Request messages; storing and forwarding the Reply message for every Request message that arrives in the configured period of time.

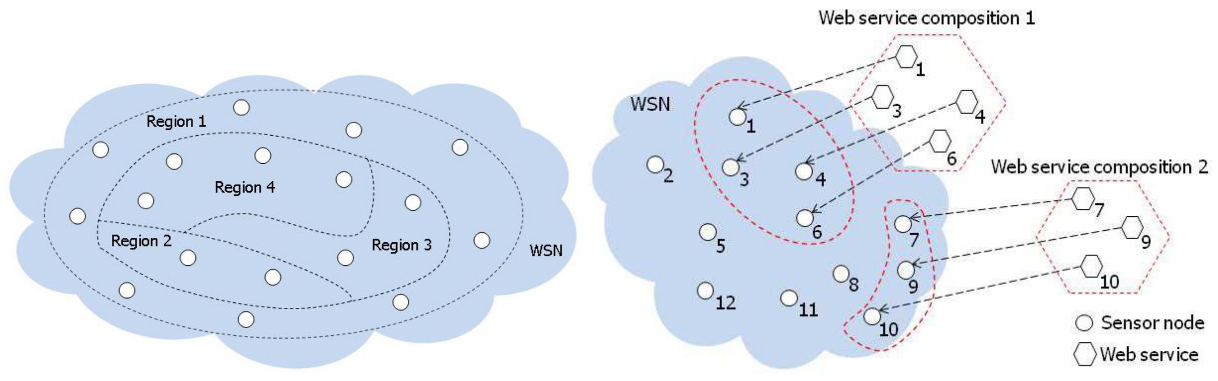
The Automatic Type Conversion is also performed by SAGe. When converting a WIOP_i Request message into a WIOP_s one, it tries to fit each argument value in the Request body in a compatible type that uses less bytes. The same procedure is performed for the result value of WIOP_i Reply messages when converting them into WIOP_s Reply messages.

Besides its active participation in those mechanisms, SAGe also performs small actions that help saving the sensor nodes energy. An example has already been described in Section 5.2.2: SAGe does not forward to WSN any Internet Request which asks for a sensor service that has not been registered. It would be useless and energy wasting since no sensor announced that service. Another example occurs when a sensor requests an Internet service (Section 5.2.3). It consists in not giving up at the first unsuccessful attempt to connect to the Internet service provider. Considering that it may be a sporadic problem, SAGe tries to connect to the server a configurable number of times before returning an error to the sensor node. This procedure aims to refrain the sensor from sending another Request in case the answer is fundamental for its application. Details of SAGe implementation are given in [10].

6. Clever

Several applications require particular attention to specific parts (regions) of a WSN. For instance, monitoring applications like a security surveillance may define the border region as a critical one to be monitored, whereas forest fire detecting applications may define clearings that resulted from previous fires as the critical regions to be monitored (since they tend to be more propitious for new fires due to their dry vegetation and soil) [25]. For this purpose, the WSN may be divided into logical regions, as illustrated in Figure 11(a).

Considering the given examples, Region 1 would be defined as the critical area for the security surveillance application and Region 4 would represent a clearing and be the critical area of the forest fire detection application.



(a) Logical regions in a WSN (b) Service composition defining regions in a WSN

Figure 11. Defining logical regions in a WSN

With that in mind, Clever is added to the framework as a tool that allows defining these logical regions by composing Web services available in the Internet. Although the notion of logical regions have already been presented in the literature, to the best of our knowledge this is the first approach that uses web service composition to define logical regions in WSNs.

It is worth mentioning that regions intersection may occur since a single web service (which represents a sensor node) may participate in more than one composition. The next section describes the composition approach used by Clever.

6.1. Composition approach

Besides WISeMid services (that is, Internet's or WSN's services which use WIOP messages to interact with each other), Web services are also supported by WISeMid. In other words, WISeMid enables transparent communication among WISeMid services and Web services. That is possible by using two different kinds of proxy: one that plays the role of a Web service, allowing other Web services to access the WISeMid service it represents, and one that acts as a WISeMid service, to enable the communication between other WISeMid services and the Web service it is related to. Both kinds of proxies are automatically generated by ProxiesGen from WIDL and WSDL files and have the same operations that the services they represent.

Considering that, each WSN node may provide a (WISeMid) service, which may be represented by a Web service that acts as a proxy. Therefore, composing Web services which represent services that are running on sensor nodes can result in the creation of a logical region in the WSN. Figure 11(b) illustrates two Web service compositions that define two logical regions in a WSN. On both compositions, when an operation (e.g., getTemperature) belonging to the interface of the Web service composition is invoked, the invocation is passed to all sensor nodes belonging to the WSN's logical region and represented by the composition.

As explained above, the Web services depicted in Figure 11(b) are actually proxies to WISeMid services running in the sensor nodes. Thus, the Web services that constitute the composition (which is itself a Web service) do not access directly the sensor nodes. Rather, they convert and forward the received requests to the associated WISeMid service provided by the nodes.

When many sensors provide the same (WISeMid) service, only one Web service is used as a proxy to them. That happens because the Web service proxy is automatically generated by the ProxiesGen based on the service's WIDL, which is identical for all sensors providing the same service. Hence, to create a logical region with sensors that provide the same service, a user must add that service `n` times, where `n` is the number of sensors that will comprise the region.

To identify to which sensor a request needs to be sent, a parameter is set in the URL of the Web service. For example, if the URL of a temperature Web service is

`http://localhost/jaxws-Temperature/Temperature`

then the resulting parameterized URL is

`http://localhost/jaxws-Temperature/Temperature?id=1` or

`http://localhost/jaxws-Temperature/Temperature?id=2`

where the id parameter is the identifier of the WSN node that provides the service.

The process of defining a WSN logical region by creating a Web service composition comprises three steps: (1) to choose the Web services to be used in the composition; (2) to set the order and the parameters for each Web service invocation; and (3) to set the URL parameter (node id) of each web service invocation. The first two steps are commonly used in Web service composition and are typically performed using BPEL and a modeling and verification tool that supports it. The third step requires modifying the WSDL file that results from the composition to add a partner link for each sensor that is supposed to compose the logical region, as explained in the following.

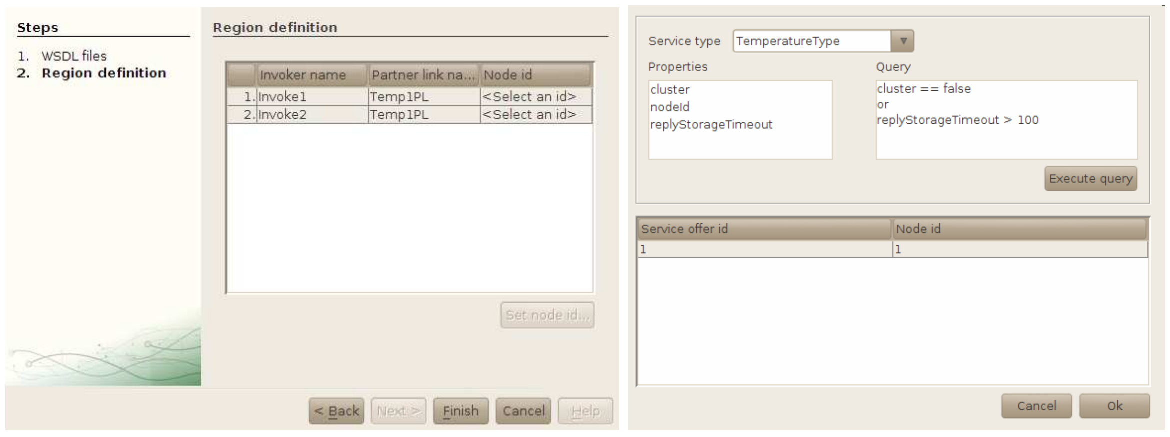
The BPEL partner links are the Web services used in the composition. Their URLs are static by default and obtained from the respective WSDL file. However, in the case that a Web service represents more than one WSN service, it is necessary to change these URLs updating dynamically the sensor identification number parameter, as there will be only one web service for them. This process is called Dynamic Addressing and an example of code is shown below:

```
<assign name="SetupPartnerlink"><copy>
 <from><literal><EndpointReference xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing"
       xmlns:tns="http://stub.mid/">
  <Address>http://localhost:8081/jaxws-Temp/Temp?id=6</Address>
  <ServiceName PortName="TempServicePort">tns:TempServiceWS</ServiceName>
 </EndpointReference></literal></from>
 <to variable="partnerReference"></to>
</copy></assign>
<assign name="AssignPL"><copy>
  <from>bpws:doXslTransform('urn:stylesheets:wrap2serviceref.xsl', $partnerReference)</from>
 <to partnerLink="Temp1PL"/>
</copy></assign>
```

```
12:<invoke name="Invoke1" partnerLink="Temp1PL" operation="getTemp" xmlns:tns="http://stub.mid/"
          portType="tns:TempService" inputVariable="GetTempIn" outputVariable="GetTempOut"/>
```

First, the new URL is assigned to an EndPointReference variable (1-7). Then, this variable is copied to a partner link (8-11). Finally, the invocation to the updated partner link is done (12).

The Clever tool is responsible for automating the third step by allowing the user to choose the nodes that should be part of the region and then automatically creating a partner link for each selected sensor node. Clever is implemented as a NetBeans plug-in, being accessed through a button. When that button is pressed, a wizard is opened to guide the user in choosing the nodes that will join the logical region. First, the Web service that represents the desired WSN service must be selected from a list of the services that have been added to the composition. Actually, the listed names are the partner link names of the invokes contained in the composition, with their respective WSDL file location. Once the Web service (partner link) is selected, the next window shows all the invokes to that service, as illustrated in Figure 12(a). For each invoke, a node ID must be set. To choose the node of interest, the user may search the WSN for all nodes that satisfy certain constraints, which are represented by property values of the offered service. For that purpose, Clever provides an interface that communicates with the Trader, as depicted in Figure 12(b). Using this window, the user is able to perform a query in the Trader, receiving a list of the nodes that satisfy that query to choose one of them.



	Invoker name	Partner link na...	Node id
1.	Invoke1	Temp1PL	<Select an id>
2.	Invoke2	Temp1PL	<Select an id>

```
cluster == false
or
replyStorageTimeout > 100
```

Service offer id	Node id
1	1

(a) Step 2 for defining a logical region: selecting the nodes (b) Finding and setting the node ID

Figure 12. Clever interface

When every invoke has a node associated, Clever automatically updates the BPEL file (of the Web service composition) to change the URL parameter before each partner link invocation. After that, the new (composite) Web service is ready and the related WSN logical region is created. Once the Web service composition is designed and deployed, a new Web service is created representing that composition. An invocation to this service corresponds to invoking each Web service involved in the composition and those Web services in turn request the related WSN service provided by each selected sensor node.

It is worth mentioning that, although the approach focuses on creating compositions with WSN Web services (more specifically, Web services representing WSN services) in order to form the logical regions in the WSN, it is possible to include Internet Web services in the composition, creating heterogeneous compositions. This possibility raises the level of integration since the (composite) service itself is distributed among nodes of both networks.

7. Energy consumption evaluation

This section presents results of experiments that analyze how WISeMid (including SAGe and the energy-saving mechanisms) and Clever affects the energy consumption in a sensor node.

For all scenarios, we use one Iris mote connected to a MTS400 basic environment sensor board, running the application defined in the studied scenario; and one Iris mote connected to a MIB520 USB programming board, working as a base station (BS), i.e., the sink node. The BS is connected to an Internet host that runs SAGe. Also, two other services run on an Internet host: the Naming service and the service/application under evaluation.

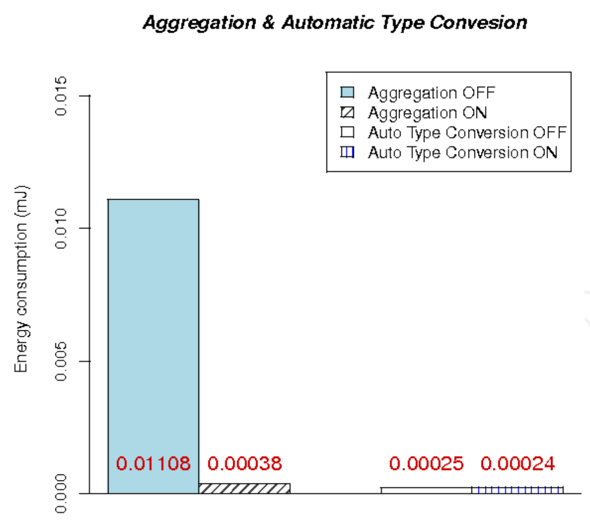
To estimate the energy consumption of the sensor node, an oscilloscope (Agilent DSO03202A) has been used. A PC is connected to the oscilloscope that captures the code snippet execution start and end times by monitoring a led of the sensor, which is turned on/off to signalize execution start/end. The PC runs a tool called AMALGHMA [26], which is responsible for calculating the energy consumption. In order to make the results more reliable, all values presented here are actually a mean value of 100 executions of the code in study.

Previous evaluation results have shown that WISeMid infrastructure has an impact over the energy consumption. Comparing an application that uses WISeMid infrastructure to a similar application that uses only TinyOS, the results shows that the service abstraction provided by WISeMid brings an energy consumption increase of 16.11% compared to the TinyOS application version (see [8] for details). Although it is not a negligible increase, the facilities offered by the WISeMid service abstraction as well as the energy saving brought by the energy aware mechanisms, which are presented next, compensate that.

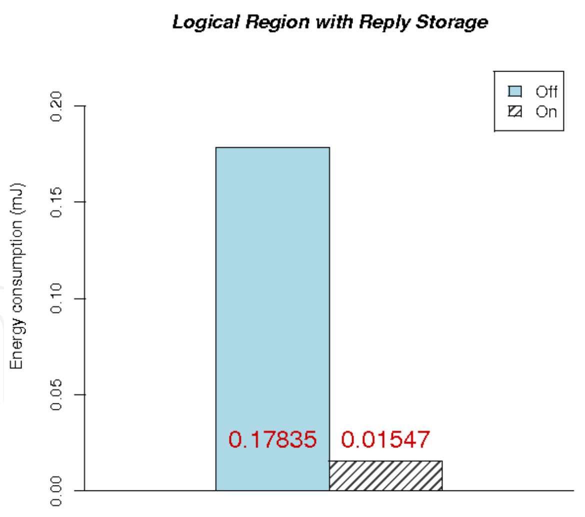
Aggregation: To analyze the energy-saving offered by the Aggregation service, a scenario composed by an Internet service user requesting a WSN service that provides the sensed temperature is studied. When the Aggregation service is off, the Internet service sends 30 requests in a row; when it is on, only one request is sent by the Internet service user, and the WSN service provider returns one value which corresponds to the aggregation (average) of the last 30 sensed values. As Figure 13(a) shows, the Aggregation service saves 96.57% of energy. That significative rate is due to avoiding the transmission of 58 messages (29 requests and 29 replies), which would be necessary if the Aggregation service was not available.

Automatic Type Conversion: The evaluation scenario for the Automatic Type Conversion comprises a simple WSN service, which has an operation that sets an attribute of type `long` (an 8-byte integer) and returns an acknowledgement, and an Internet service user that requests this operation with value "1" (one). As this value fits in type `byte`, when the Automatic Type Conversion feature is on, SAGe converts the parameter type from `long` to `byte` and only one byte is used to transmit the value "1", instead of 8 bytes. The results are illustrated in Figure 13(a) and show that, for the described scenario, the energy-saving gain is 4% when the Automatic Type Conversion is performed. To confirm whether that decrease in energy consumption is statistically significant, we performed a Paired t-test with those measurements. The resulting p-value is less than 4.352e-05, indicating the means are really different. Although 4% is not a very expressive gain, it is still an important gain as every little bit of energy that is saved in a WSN contributes to preserve and extend its lifetime.

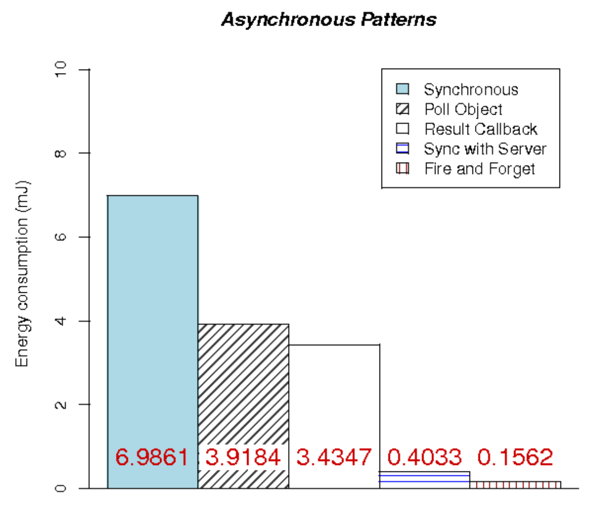
Reply Storage Timeout in Logical Regions: The next scenario involves the use of a WSN logical region composed by two sensor nodes providing a Temperature service, which may also be invoked as a web service called TemperatureWS. As explained previously, only one proxy



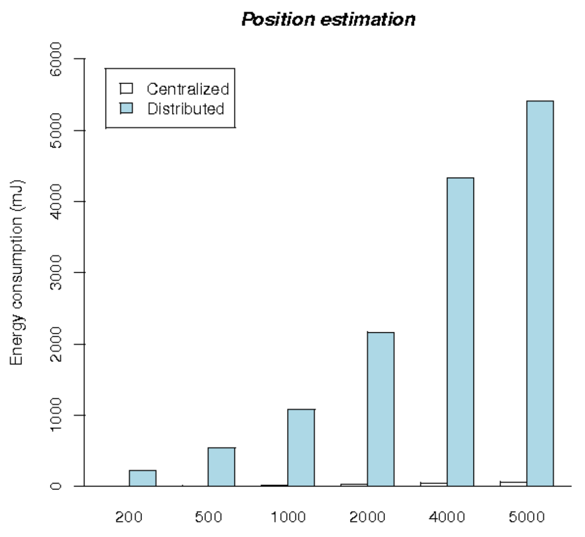
(a) Aggregation service and Automatic Type Conversion



(b) Reply Storage Timeout used in Logical Regions



(c) Asynchronous Patterns instead of Synchronous communication



(d) Sensor position estimation method being performed by the WSN or by the Internet

Figure 13. Energy consumption of energy-saving mechanisms and real application

represents all sensors that provide the same service, hence TemperatureWS represents both sensors' service. A Web service composition (WSComposition) with the two Temperature services (actually, the TemperatureWS proxy) is created using Clever for selecting the nodes of interest. Finally, a service that runs on the Internet (TempMon) and monitors the temperature in that area calls WSComposition, which actually calls TemperatureWS twice in a row, once for each sensor node. The proxy calls the Temperature service in the sensors and then forwards their replies to the WSComposition, which calculates the mean value of the received results and returns it to the TempMon service.

To evaluate this logical region composition with the Reply Storage feature, the TempMon service requests the Temperature service composition (WSComposition) 25 consecutive times with random interval between messages, and a timeout of 4s. Figure 13(b) presents the results, which show that the Reply Storage feature saves 91.28% of energy in the studied scenario, which is a very significant energy-saving rate.

Asynchronous invocation patterns: The asynchronous patterns are evaluated in groups according to their type of operations: one-way or request-response. The one-way scenario is composed by a Logging service provided by an Internet host that is used by a WSN service to record log messages. The patterns evaluated here are Fire and Forget and Sync with Server.

The request-response scenario consists of the first distributed application described in Section 2, with a Surveillance service running in a sensor node, which captures images of any moving target and sends them to a Recognition service running in the Internet.

The results of both scenarios are presented in Figure 13(c). For one-way operations, a sensor saves 61.27% of energy when Fire and Forget is adopted instead of Sync with Server. That significant difference was expected since the former ends the process the moment the invocation is sent, whereas the latter involves the tasks performed in SAGe (such as message translation) plus the round-trip delay time from SAGe to the server. For response-reply operations, comparing to synchronous communication, the Poll Object pattern saves 43.91% of energy whilst the Result Callback obtains 50.83% of energy-saving. Those results confirm that the asynchronous invocation patterns can be adopted to extend the WSN lifetime.

Requesting Internet Services: One of WISeMid's main characteristics is that it allows sensor nodes to be not only service providers, but also service users. There are several situations in which asking for a node outside the network to perform a task is more suitable than performing it in-network. It may be even less costly in terms of energy consumption to request the service outside the WSN. Next scenario evaluates an environment monitoring application that shows this strategy. This application is composed by a Monitoring service that runs in a mobile sensor node and therefore needs to know the node position before reporting measured data (see [27] for examples of real applications with mobile sensor nodes). The Monitoring service may estimate the node position or it may request the position estimation from another service (called Positioning), which is provided by an Internet node. This scenario evaluates the energy consumption of the sensor node that provides the Monitoring service in both situations: performing and requesting the node position estimation.

To estimate the node position, we adopted an approach that has been developed with two versions: centralized and distributed. Both versions use measurements of distance between every node and its neighbors. In the centralized version, all sensor nodes send information about their neighbors to a central machine (outside WSN) with plenty of computation power where the nodes position are calculated and sent back to the network. In the distributed version, each node is responsible for calculating its position using information about its neighbors. Details of this localization approach, including implementation description and accuracy evaluation results, can be found in [28, 29].

Considering those information, we were able to estimate the energy consumption for both versions of this approach. The centralized version consumption comprises the energy that a node spends by requesting its location to a service that is provided by the Internet (Positioning service). It is worth mentioning that synchronous communication is being used, therefore this energy consumption includes the time the sensor keeps waiting for the reply. Moreover, to perform the localization algorithm, the Positioning service must have received the neighbor information of all WSN nodes. Hence, we have added to this calculation the energy consumption of all sensor nodes sending a message with their neighbors information.

For the distributed version, the values represent the estimated energy consumption of all messages that have to be exchanged for the position calculation. That value actually comprises the estimation of the number of messages that each node sends during all the localization algorithm execution, which involves several iterations in the optimization process [29] and different types of messages [28]. This estimated number of messages was used to measure the power a sensor node consumes to send all those messages, then this measured value was multiplied by the number of sensor nodes that compose the WSN.

Figure 13(d) shows the estimated energy consumption for both versions considering different sizes of network. As one can observe, in some situations, asking for a node outside the network to perform a task is more suitable than performing it in-network. For this specific application which uses this localization algorithm, the centralized version consumes approximately 98% less energy than the distributed version. That huge energy-saving rate is due to the great number of messages that have to be exchanged by the sensor nodes in the distributed version. In a network with 200 nodes, for instance, approximately 22,400 messages are transmitted.

8. Concluding remarks

This chapter introduced a framework for integrating WSNs and the Internet at service level, by allowing the interoperability of their services. The framework is composed by four components: Smart, a service model to describe services from both networks (Internet and WSNs); WISeMid, a middleware that enables the interoperability of WSN's and Internet's services; SAGe, an advanced gateway which is responsible for providing the location and access transparency for the services' communication; and Clever, a service composition tool which enables the creation of logical regions in the WSN by just composing Web services that are available on the Internet.

Some evaluation results concerning the energy consumption of those elements have been presented. Those results lead to the conclusion that by using some WISeMid's energy-saving mechanisms, a sensor node may save significant amount of energy. For instance, using the Aggregation service, a sensor node can save 96.57% of energy, whereas by using the Reply Storage Timeout feature in a logical region, a saving of 91.28% is obtained. Furthermore, a real application which requests an Internet service that estimates the sensor positioning instead of computing it in-network saved approximately 98% of energy, emphasizing the usefulness of the sensor nodes' capability of being service requestors.

In terms of future work, there are a number of possibilities to improve and extend the framework elements. The Smart model may be improved by detailing some elements, such as `MessageExchangePattern`'s `Publish/Subscribe`, and those elements may be implemented in WISeMid. Some features may also be added to SAGe, such as turning it into a distributed service, to refrain it from becoming a bottleneck in large-scale WSN. Also, some energy-saving mechanisms may be improved. For instance, the Reply Storage Timeout feature current implementation, which uses a global and fixed timeout value, would probably increase its energy gains by implementing an adaptive method to automatically update the timeout value per service, based on the variation history of the data sensed by the WSN nodes.

Author details

Jeisa Domingues
Federal Rural University of Pernambuco, Brazil

Antonio Damaso, Rilter Nascimento and Nelson Rosa
Federal University of Pernambuco, Brazil

9. References

[1] Rao S G., Zeldy S., Usman S., Mazlan A. A gateway solution for IPv6 wireless sensor networks. In: proceedings of the International Conference on Ultra Modern Telecommunications & Workshops (ICUMT '09), pp. 1-6; 2009.

[2] Smolderen K., De Cleyn P., Blondia C. Wireless Sensor Network Interconnection Protocol. In: proceedings of the IEEE Globecom 2010 Workshop on Heterogeneous, Multi-hop Wireless and Mobile Networks (HeterWMN 2010), pp. 164-168; 2010.

[3] Yoon I-S., Chung S-H., Jeong-Soo K. Implementation of Lightweight TCP/IP for Small, Wireless Embedded Systems. In: proceedings of the IEEE 23th International Conference on Advanced Information Networking and Applications (AINA), pp. 965-970; 2009.

[4] Bai J., Zang C., Wang T., Yu H. A Mobile Agents-Based Real-time Mechanism for Wireless Sensor Network Access on the Internet. In: proceedings of the 2006 IEEE International Conference on Information Acquisition, pp. 311-315; 2006.

[5] Samaras IK., Gialelis JV., Hassapis, GD. Integrating Wireless Sensor Networks into Enterprise Information Systems by Using Web Services. In: proceedings of the Third International Conference on Sensor Technologies and Applications (SENSORCOMM '09), pp. 580-587; 2009.

[6] Priyantha NB., Kansal A., Goraczko M., Zhao F. Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. In: proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, pp. 253-266; 2008.

[7] Domingues J., Damaso A., Rosa N. Smart: Service Model for Integrating Wireless Sensor Networks and the Internet. In: proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS), pp. 365-372; 2010.

[8] Domingues J., Damaso A., Rosa N. WISeMid: Middleware for Integrating Wireless Sensor Networks and the Internet. In: proceedings of the 10th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS), pp. 70-83; 2010.

[9] Domingues J., Damaso A., Nascimento R., Rosa N. An Energy-aware Middleware for Integrating Wireless Sensor Networks and the Internet. International Journal of Distributed Sensor Networks 2011; Article ID 672313, 19 pages.

[10] Damaso A., Domingues J., Rosa, N. SAGe: Sensor Advanced Gateway for Integrating Wireless Sensor Networks and Internet. In: proceedings of the IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 698-703; 2009.

[11] Estefan J., Laskey K., McCabe F., Thornton, D. OASIS Reference Architecture for Service Oriented Architecture (SOA-RM), Version 1.0; 2008.

[12] Racz P., Stiller, B. A Service Model and Architecture in Support of IP Service Accounting. In: proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), pp. 1-12; 2006.

[13] Gracanin D., Eltoweissy M., Wadaa A., DaSilva L. A service-centric model for wireless sensor networks. IEEE Journal on Selected Areas in Communications 2005; 23 (6): 1159-1166.

[14] Rezgui A., Eltoweissy M. Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead. Computer Communications 2007; 30 (13): 2627-2648.

[15] Völter M., Kircher M., Zdun U. Foundations of Enterprise, Internet, and Real-Time Distributed Object Middleware. New York: John Wiley & Sons, Inc.; 2004.

[16] Rubio B., Diaz M., Troya JM. Programming Approaches and Challenges for Wireless Sensor Networks. In: proceedings of the 2nd International Conference on Systems and Networks Communications, pp. 36-43; 2007.

[17] OMG. Trading Object Service Specification. OMG Technical Document Number 00-06-27; 2000.

[18] Benigni F., Brogi A., Fuentes T., Popescu R. D2.1: Service Model Description. SMEPP Consortium; 2008.

[19] Ran S. A model for web services discovery with QoS. SIGecom Exch. 2003; 4 (1): 1-10.

[20] Pokraev SV., Quartel, DAC., Steen MWA., Reichert MU. A Method for Formal Verification of Service Interoperability. In: proceedings of the IEEE International Conference on Web Services (ICWS'06), pp. 18-22; 2006.

[21] Sahai A., Singhal S., Machiraju V., Joshi R. Automated policy-based resource construction in utility computing environments. In: proceedings of the 2004 IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), pp. 381-393; 2004.

[22] Deschrevel J-P. The ANSA Model for Trading and Federation. ANSA Architecture Report APM.1005.01; 1993.

[23] Schantz R., Schmidt D. Middleware for Distributed Systems: Evolving the Common Structure for Network-Centric Applications. In: Encyclopedia of Software Engineering, J. Marciniak, and G. Telecki, Eds. John Wiley & Sons, Inc., New York; 2002.

[24] Levis P., Gay D. TinyOS Programming. New York: Cambridge University Press; 2009.

[25] Carvalho Jr. JA., Veras CAG., Alvarado EC., Sandberg DV., Carvalho ER., Gielow R., Santos JC. Fire spread around a forest clearing site in the Brazilian Amazon Region. In: proceedings of the 2nd International Wildland Fire Ecology and Fire Management Congress; 2003.

[26] Tavares E., Silva B., Maciel P. An Environment for Measuring and Scheduling Time-Critical Embedded Systems with Energy Constraints. In: proceedings of the IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 291-300; 2008.

[27] Romer K., Mattern F. The Design Space of Wireless Sensor Networks. IEEE Wireless Communications 2004: 11, 54-61.

[28] Marks M., Niewiadomska-Szynkiewicz E. Multiobjective Approach to Localization in Wireless Sensor Networks. Journal of Telecommunications and Information Technology 2009: 3, 59-67.

[29] Niewiadomska-Szynkiewicz E., Marks M. Optimization schemes for wireless sensor network localization. Int. J. Appl. Math. Comput. Sci. 2009: 19, 291-302.