Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2011, Article ID 672313, 19 pages
doi:10.1155/2011/672313

Research Article

An Energy-Aware Middleware for Integrating Wireless Sensor Networks and the Internet

Jeisa Domingues, Antonio Damaso, Rilter Nascimento, and Nelson Rosa

Informatics Center, Federal University of Pernambuco, P.O. Box 7851, 50740-540 Recife, PE, Brazil

Correspondence should be addressed to Jeisa Domingues, jpo@cin.ufpe.br

Received 1 September 2010; Revised 29 December 2010; Accepted 7 February 2011

Wireless sensor networks (WSNs) have great potential for many distributed applications in different scenarios. As they grow in popularity and importance, it becomes increasingly desirable and necessary to share their data over the Internet. This paper presents an energy-aware middleware that focuses on integrating the Internet and WSNs at service level by providing transparency of access, location, and technology. In particular, it describes and evaluates the implemented strategies to reduce the energy consumption, which are essential for prolonging the WSN lifetime.

1. Introduction

Wireless sensor networks (WSNs) have great potential for many applications in scenarios such as military target tracking and surveillance, natural disaster relief, biomedical health monitoring, hazardous environment exploration, and seismic sensing. For that reason, they have received increasing attention over the past few years. As WSNs become more numerous and their data more valuable, it becomes increasingly important to have common means to share data over the Internet [1].

Since WSNs can be easily deployed in various environments to monitor target objects and various conditions and to collect information, they are considered one essential infrastructure for pervasive computing systems. Sensor nodes can be embedded in walls, chairs, watches or stick on the cups, bags, and so on, to gather users' information. These data can be crucial to pervasive computing applications, as they provide the fundamental information of users' context and the underlying execution environment [2]. Also, the WSN is one of the many networks that will compose the Ambient Networks [3]. For all those reasons, integrating WSN with the Internet has become increasingly desirable and necessary.

A number of solutions have been proposed in recent years to allow the WSN and Internet integration [4, 5, 16–25]. Most of them aim at integrating those networks through mapping protocol stacks and logical address formats used in both networks. Those solutions focus on accessing the network nodes through their logical addresses, which raises several problems.

In this context, this paper extends a solution that aims at integrating applications instead of networks (i.e., protocols stack and/or logical address formats mapping) by incorporating strategies to reduce the WSN power consumption. The proposed solution is an energy-aware middleware, namely, WISeMid, that allows integrating applications, which are considered services, in a transparent way. The unique contributions of this paper are the proposed power-saving mechanisms: 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 considered up-to-date; Automatic Type Conversion, which removes unnecessary bytes from the messages; the implementation of invocation asynchrony patterns, which prevents the sensor application from wasting power for being blocked during a service requesting. In order to analyze the impact of the proposed strategies, we also present a power consumption evaluation in an actual application.

The remainder of this paper is organized as follows. Section 2 outlines details of WISeMid's main characteristics, architecture, and implementation. Next, Section 3 describes the power-saving strategies implemented in WISeMid.

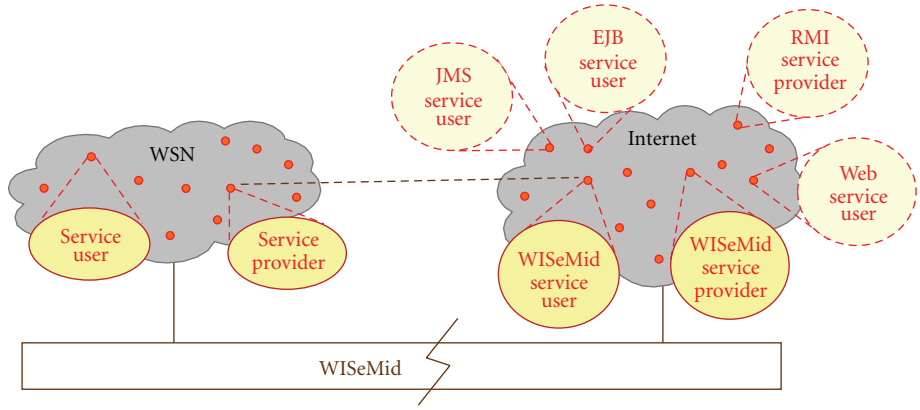

Figure 1: WISeMid logical view.

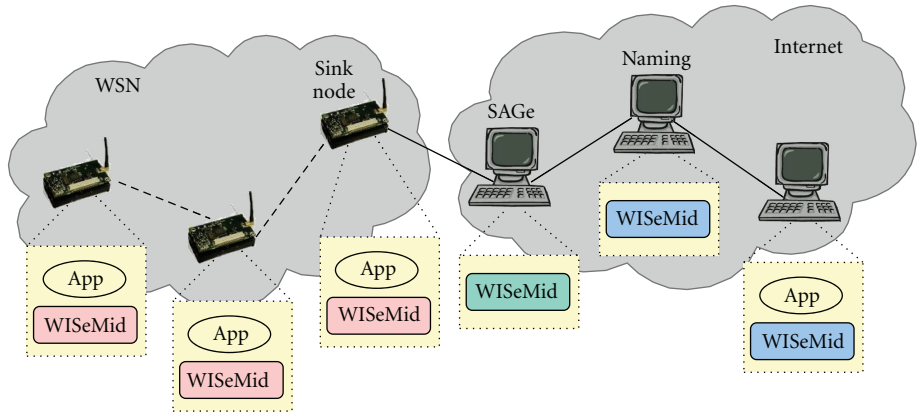

Figure 2: WISeMid physical view.

Section 4 presents the analysis of the power consumption of the proposed strategies. Section 5 presents related work. Finally, Section 6 draws our conclusions and identifies future work.

2. WISeMid

WISeMid (Wireless sensor network's and Internet's Services integration Middleware) [6] is a communication infrastructure that supports the integration of WSNs and the Internet at service level (see Figure 1). In this context, applications running in the Internet/WSN nodes may play the role of service providers or service users. In practice, a service user must be able to communicate with a service provider no matter whether they are running in the same network or not. Hence, WISeMid should provide an infrastructure that allows integrating these services in such a transparent manner that a service should be accessed in the same way irrespective of being provided by a WSN node or by an Internet host. Additionally, WISeMid should support application services developed in different technologies, such as Java RMI, EJB and Web Services, although its current implementation supports only WISeMid services and Web Services.

Figure 2 presents a physical view of how WISeMid spreads out through the Internet and WSN. The WISeMid interface for application developers is exactly the same, but the implementations for WSN and Internet are different, as they have distinct requirements and components. The 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 named SAGe, which acts as a proxy between both networks (see Section 2.5).

2.1. Overview. In order to promote the networks integration at service level, our middleware has to address some issues. WISeMid should treat four different kinds of heterogeneity, namely, operating system, network, hardware, and programming language. WSN and Internet nodes have different hardware platforms (e.g., PC and MicaZ nodes) and network protocol stack (e.g., ZigBee and TCP/IP). In addition, applications running in both networks are developed atop different operating systems (e.g., Windows or Unix-based OS, and TinyOS) and using distinct programming languages (e.g., Java and nesC).

The heterogeneity of programming language raises another issue: data type mapping. For example, consider that

```
(1) module PACKAGE_NAME {
(2)    interface INTERFACE_NAME {
(3)       [OPER_TYPE] OUTCOME_TYPE OPER_NAME (TYPE ARG1, ...])
                                          [raises (EXCEPTION_NAME1, ...)]
(4) }    }
```

Algorithm 1

a service user is written in nesC (it runs in the WSN) and a service provider is written in Java (it runs in the Internet). When the service user invokes an operation in the service provider, it is necessary to translate nesC data types into Java ones in a transparent way to application developers.

Along with handling the considered heterogeneities, it is also necessary to define a notation for describing services (e.g., an Interface Definition Language), the basic abstractions used to build distributed applications (e.g., objects, services), the communicating entities (e.g., client/servers, peers), the way these entities communicate (e.g., synchronously, asynchronously), and distributed services provided by the middleware (e.g., naming service, aggregation service).

2.2. WISeMid IDL. As service is the key concept in the proposed approach, it is necessary to propose a notation to be used to describe services. 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, that is, wherever the service runs (Internet or WSN) and whatever the implementation language (Java or nesC), the service's interface is described using a unique language. The general structure of an interface defined in WISeMid IDL is shown in Algorithm 1.

The module (package) that contains the service should be initially specified (1). Then, the service interface includes its name (2) and provided operations (3). Each operation has a name, input/output typed parameters and may raise exceptions. Additionally, an operation is by default a request-response operation, but it may be defined as a one-way operation, that is, no response is expected when the operation is invoked.

2.3. Requirements. Considering the issues introduced in Section 2.1, the following WISeMid's requirements have been identified:

(R01) service providers should register their services in registry;

(R02) service users should access the registry for the service it wants to use;

(R03) service users are not aware of the location of the service being used (location transparency);

(R04) service users access local and remote services in a similar way (access transparency);

(R05) service data should have the same interpretation whatever the programming languages used to implement the service users and service providers;

(R06) service providers/users communicate among themselves using Request/Reply communication pattern;

(R07) service communication is synchronous; (R08) services should be stateless and untyped for simplicity.

In addition to these requirements, which concern both networks, there are some specific requirements regarding the limited resources of sensor nodes: (R09) the messages transmitted to the WSN should be kept as short as possible (e.g., if an argument type uses four bytes to represent a value but the argument value fits in one byte and there is a compatible type that uses only one byte to represent that value, our middleware should convert this argument to the smaller type before sending it to a sensor node); (R10) unnecessary messages should not be forwarded to the WSN (e.g., when an Internet application requests a sensed data like temperature, whose variability considering second/minute time scale is not so significative, WISeMid may decide not to forward this request to the WSN, returning to the application the last value obtained from the WSN).

2.4. Architecture. The WISeMid architecture is depicted in Figure 3 and consists of three layers: Infrastructure, Distribution, and Common Services.

The Common Services layer includes services that are not particular to a specific application domain: Aggregation, which performs sensor data aggregation (it runs in the WSN); Grouping, which defines clusters inside the WSN; Naming, that stores information needed to access a service (it runs in the Internet); SAGe, that is in charge of forwarding messages from/to WSN (it runs in the Internet). Additionally, SAGe also provides location transparency acting as a service proxy between both networks and performs some tasks concerning the aforementioned WSN-specific requirements (R09 and R10) in order to avoid waste of sensors restricted resources (see Section 2.5).

The Distribution layer includes the following elements: the Stub, which represents a local instance of the service within the client process and offers the same interface as the remote service; the Requestor, which constructs a remote invocation on the client side from parameters such as remote service location, service name, and arguments; the Skeleton, which dispatches remote invocations to the remote service using the invocation information sent by the Requestor; the Marshaller, which serializes and deserializes

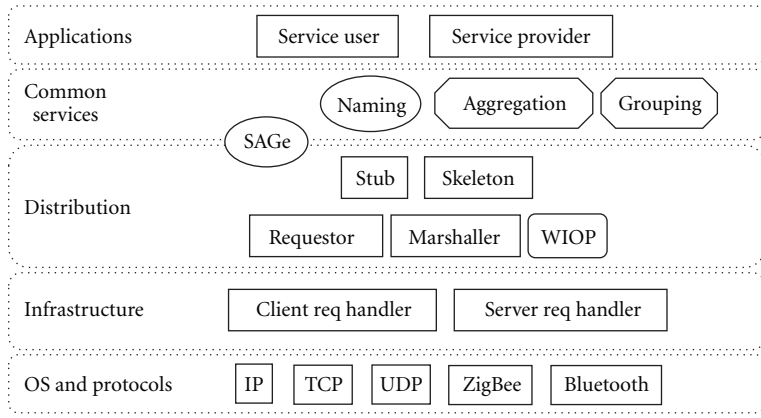

Figure 3: WISeMid architecture.

the parameters passed between client and server using the WIOP Messages. WIOP is our middleware interoperability protocol, which defines the Request/Reply messages between client and server.

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, for example, sockets (Windows) and ActiveMessageC (TinyOS).

As sensor nodes have limited resources, some elements of the WISeMid architecture 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, which means that the Marshaller is not necessary.

2.5. Implementation. The WISeMid implementation is divided into two parts, one for the WSN nodes, developed in nesC, and another for the Internet hosts, developed in Java. Implementation details of the main middleware elements are described as follows.

2.5.1. WIOP. 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 4. The WIOP message header is composed of the following fields: `endianness` (e.g., big endian or little endian); `msgType` (e.g., request or reply message); `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 (i.e., 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 5.

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; `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_i). The version that runs in the WSN (called WIOP_s) contains three additional fields (that are emphasized in Figure 5): `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 (meeting the (R09) requirement). 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 [7]. In such cases, the data is divided into 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, with each argument being composed of 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 (e.g., the `float` type is represented by the number 7), 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. In this case, three arguments

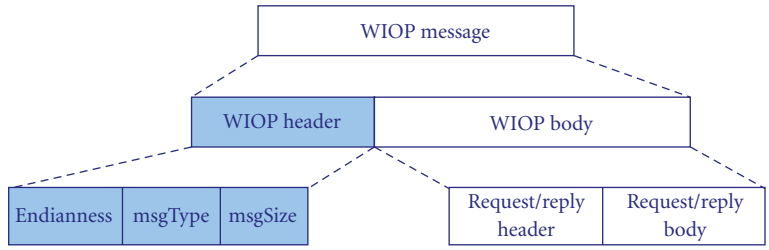

Figure 4: WIOP message header.

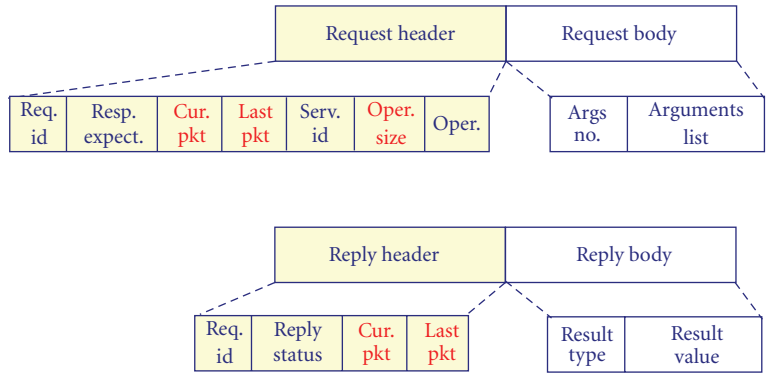

Figure 5: WIOP message body.

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; `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). Analogously to the Request message header, the Reply message header in the WSN version also contains the additional fields related to the message fragmentation/reassembly: `currentPacket` and `lastPacket`.

The Reply body is composed of the result type and its value for both formats: the Internet and WSN (see Figure 5).

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

The WISeMid Naming Service and Internet services use the Internet format ($WIOP_i$), while the sensor services use the WSN format ($WIOP_s$). Only SAGe handles both formats.

2.5.2. Naming Service. The WISeMid Naming Service stores the references of services executing in the Internet and WSN in such 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; List, to list all registered services. The service reference includes the service ID, endianness, the IP address, and the port number. In the case the service is running in the WSN (i.e., the sensor node has not an IP address), the stored IP address is the SAGe address, as described next.

2.5.3. SAGe. As stated in Section 2.4, SAGe is an important element in WISeMid architecture. Running in the Internet host connected to the WSN sink node via a serial port (see Figure 2), SAGe's main function is to act as a service proxy between both networks by enabling the communication between services running on the Internet hosts and WSN nodes in a transparent way. Furthermore, SAGe also performs some tasks concerning the previously defined requirements that are specific to WSN: (R09) and (R10). The rest of this section describes how SAGe provides the location transparency and implements the WSN requirements.

(a) 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.

(b) Invocation of a WSN Service. When the SAGe receives a ($WIOP_i$) 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 was 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 (avoiding the transmission of unnecessary messages as stated in the (R10) requirement). Instead, it sends a Reply message reporting an error to the Internet host that requested the service.

(c) 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, that is, 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.

3. Proposed Approaches for Energy-Saving in Middleware for WSN

As energy is crucial and scarce resource in WSNs, performing energy-saving methods is essential to extend the sensor nodes lifetime and thus the WSN lifetime. This section proposes approaches for conserving energy in middleware for WSN.

3.1. Aggregation. Communication in wireless sensor networks is very expensive in terms of energy because a sensor node spends most of its energy in data transmission and reception. For that reason, techniques have been developed to reduce overall data communication, such as data aggregation.

Data aggregation consists of aggregating redundant or correlated data in order to reduce the overall size of sent data, thus decreasing the network traffic and energy consumption [8]. To perform the data aggregation, a function is applied to all sensors data. This aggregation function may be simple, such as average, mode, sum, minimum, and maximum or it may be more complex, like approximate contours of nodes' residual energy or approximate the boundary line between sensing and no-sensing (e.g., of light) with a straight line or a parabola (for details, see [9]).

Besides this in-network data aggregation, which is used by most middleware, we propose 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.

3.2. Reply Storage Timeout. Some physical aspects sensed by a sensor node, such as temperature, do not present a great variability considering second/minute time scale. Hence, if a node is requested to sense, process, and return this kind of data very often, the returned data are likely to be the same during a short period of time.

As stated previously, avoiding unnecessary communication is an efficient manner of saving energy. With that in mind, we propose a solution that aims at preventing the transmission of redundant data. The approach consists of avoiding sending equivalent Request messages (i.e., messages asking for the same service with the same parameters) during a short period of time. Assuming that some sensed values do not change very quickly, sending the same Request for the same service in a short time period will likely return the same value, resulting in unnecessary processing and energy consumption. 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.

3.3. 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. Depending on the programming language used to develop the application, data types have a number of bytes to be represented. No matter the language, the data occupies all bytes it needs to represent the maximum number it supports. For instance, in Java, a `long` takes 8 bytes to represent a signed integer. Therefore, it has a minimum value of −9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). However, any integer value within this range will be stored in 8 bytes, no matter if only 1 byte should be enough to represent it. The value "1," for example, will have 7 bytes filled with null bits (i.e., "0") preceding (or succeeding, depending on the adopted endianness) a byte with the binary representation of the number "1."

The (R09) WSN specific requirement defined in Section 2.3 says that the messages transmitted to the WSN should be kept as short as possible. To meet this requirement, 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 (i.e., a compatible type that uses less bytes). For instance, if the argument is a `long` (an integer of 8 bytes) but its value is "525," it can be stored into a `short` (an integer of 2 bytes). Thus SAGe converts the argument from a `long` into a `short` and adds only 2 bytes to

```
(1) module MidReqHandlerM {...}
(2) implementation {...
(3)   command void ReqHandler.ffInvoker (bool repEx,
                                         uint8_t servId, char* op,
                                         uint8_t* arg, uint16_t argSize) {
(4)     if (requestId >= 63) requestId = 0;
(5)     else requestId++;
(6)     call WiopMsg.create_request_header(&req, requestId,
                                                          repEx);
(7)     call WiopMsg.setCurrentPacket (1);
(8)     call WiopMsg.setLastPacket (1);
(9)     call WiopMsg.setServiceId (servId);
(10)    call WiopMsg.setOperation (op, strlen(op));
(11)    call WiopMsg.setArguments (args, argSize);
(12)    call sender.sendMessage (&req,
                                         call WiopMsg.getMsgSize ());
(13)  } }
```

Algorithm 2

the $WIOP_s$ message instead of the original 8 bytes, avoiding the transmission of unnecessary 6 bytes. The same step is performed for the result value of $WIOP_i$ Reply messages when converting them into $WIOP_s$ Reply messages.

3.4. Invocation Asynchrony Patterns. For simplicity, we have specified that the service communication is synchronous (R07). However, synchronous invocation blocks the service user (client) until the result returned from the service provider (server) is received. When a service user is running in a sensor node, it keeps consuming power while waiting for the answer without executing any task. For that reason, we decided to extend this requirement by adding invocation asynchrony patterns, which enable the client to resume its work immediately after a remote invocation is sent. The idea is to prevent the sensor application from wasting power for being blocked during a service requesting.

Four invocation asynchrony patterns, which are presented in [10], have been implemented in WISeMid and are described as follows.

3.4.1. Fire and Forget. There are services with operations that neither have a return value nor report exceptions. For that reason, the service user (client) should not keep waiting for an answer as there will not be one.

The Fire and Forget pattern is appropriated for that kind of service. It describes one-way operations with "best effort" semantics. Once the invocation is sent across the network by the Requestor, the service user (client) is free to resume its work. Remoting errors during the sending of the invocation to the remote service, or errors that were raised during the execution of the remote invocation, cannot be reported back to the client. The client is unaware of whether the invocation is executed successfully by the remote object.

A code snippet of the RequestHandler component (called `MidReqHandlerM`) for a sensor service that adopts this pattern is presented in Algorithm 2. The code is written in nesC.

The Request Handler invoker command (3) receives the remote service ID, the operation name, and the operation arguments as parameters. After calculating the request ID (4)-(5), it creates the message (6)–(11) and sends it (12), returning the execution flow to the component who called this command.

3.4.2. Sync with Server. There are some situations when a service user (client) needs to know if its request has been received by the service provider (server), even if the requested operation neither has a return value nor reports exceptions. Sync with Server addresses these needs.

Analogously to the Fire and Forget pattern, the Sync with Server pattern also works only for one-way operations, but in this case the service user is notified about the successful delivery of the invocation to the service provider. The service user sends the invocation and waits for a reply from the service provider informing it about the successful reception (but not the execution) of the invocation. Only after that reply is received by the Requestor, the service user execution continues. The service provider executes the invocation independently after sending the acknowledgment of the invocation reception.

It is worth noting that although Sync with Server enables informing the service user of system errors (such as failed transmission of the invocation), it does not allow the service user to be informed about application errors during the remote invocation (as this happens asynchronously after the acknowledgment has already been sent to the user).

A code snippet of the RequestHandler component (called `MidReqHandlerM`) for a sensor service that adopts the Sync with Service pattern is presented in Algorithm 3. The code is written in nesC.

Analogously to the Fire and Forget pattern code presented previously, the Request Handler invoker command

```
(1) module MidReqHandlerM {...}
(2) implementation {...
(3)    command void ReqHandler.ssInvoker (bool repEx,
                                          uint8_t servId, char* op,
                                          uint8_t* arg, uint16_t argSize) {
(4)       uint8_t tryMax = 0;
(5)       if (requestId >= 63) requestId = 0;
(6)       else requestId++;
(7)       call WiopMsg.create_request_header (&req, requestId,
                                                          repEx);
(8)       call WiopMsg.setCurrentPacket (1);
(9)       call WiopMsg.setLastPacket (1);
(10)      call WiopMsg.setServiceId (servId);
(11)      call WiopMsg.setOperation (op, strlen (op));
(12)      call WiopMsg.setArguments (args, argSize);
(13)      call sender.sendMessage (&req,
                                   call WiopMsg.getMsgSize ());
(14)      while (!hasResponse (requestId)) {
(15)      call ThreadSleep.sleep (50);
(16)      if (++tryMax > 100) return (int8_t *) 0;}
(17)    getResponse (requestId);
(18) } }
```

Algorithm 3

(3) receives the remote service ID, the operation name, and the operation arguments as parameters. After calculating the request ID (5)-(6), it creates the message (7)–(12) and sends it (13). The main difference between them is that the invoker that implements the Sync with Server pattern waits for an answer (14)–(17), which is actually an acknowledgment that the message has been received by the service provider.

3.4.3. Poll Object. The Poll Object pattern is designed for request-response operations. It provides service users (clients) with the means to query a distributed object middleware about whether an asynchronous reply to the request has arrived, and if so, to obtain the return value. The Poll Object is created by the Requestor to enable the service user to resume its execution after sending the invocation, while the Poll Object keeps waiting for the service provider's reply. The Poll object then provides (at least) two operations: one to check whether the result is available and another to actually return the result to the calling client.

In order to provide a better understanding, a code snippet of a sensor application that implements the Poll Object pattern is shown in Algorithm 4.

The `POExampleM` application is used in the evaluation process (presented in Section 4). It just sequentially requests a service that is provided by the Internet (called `RECOGNITION`). The application is composed of 4 (four) events which are timers that are scheduled by the application itself. The first event (3)–(5) requests the Naming Service's lookup operation to discover the service ID of the `RECOGNITION` service (4). Then it starts the timer (5) of the second event (6)–(10), which keeps checking periodically if the reply for the Naming Service request has arrived (7). If so, it schedules the next event (10), which is responsible for requesting the desired `RECOGNITION` service's operation (12) and starting the fourth event (13). This last event just checks periodically if the reply for the request has arrived (15).

3.4.4. Result Callback. Some services need to be informed immediately if the result becomes available to the REQUESTOR, so that it can react on the availability of the result. Result Callback suits that purpose. Also used for request-response operations, this pattern allows the service user to react immediately to the results of asynchronous invocations, as it actively notifies the requesting service user of the returning result.

The major difference between Poll Object and Result Callback is that the latter requires an event-driven design, whereas the former allows large parts of the client to be written using a synchronous execution model.

To emphasize the difference between those two patterns, a code snippet of the same `RCExampleM` application now using the Result Callback pattern is presented in Algorithm 5.

Similarly to the Poll Object pattern, this application is composed of 4 (four) events. The first one (3)-(4) requests the Naming Service's lookup operation to discover the service ID of the `RECOGNITION` service (4). The second event (5)–(7) is executed when the lookup result is received by the middleware, then it schedules the next event (7), which is responsible for requesting the desired `RECOGNITION` service's operation (9). The last event (10)–(13) occurs when the result for the requested operation is received by the middleware. It performs the necessary actions depending on the result value (11)-(12) and schedules the next request of the `RECOGNITION` service (13).

```
(1) module POExampleM {...}
(2) implementation {...
(3)    event void Timer1.fired () {
(4)        call NamingService.lookup ("RECOGNITION");
(5)        call Timer2.startPeriodic (100); // Waiting for
                                                the result}
(6)    event void Timer2.fired () {
(7)        if (call PollObject.resultAvailable ()) {
               //If there is a result...
(8)            call Timer2.stop (); //... stop this timer
(9)            call Recongition.setServiceId (*((int16_t*) call
                                        PollObject.getResult ()));
(10)           call Timer3.startOneShot (80000);
               //Start sending the message } }
(11)   event void Timer3.fired () {
(12)       call Recognition.isHuman (arrImage, size);
(13)       call Timer4.startPeriodic (50); //Waiting for
                                                the result! }
(14)   event void Timer4.fired () {
(15)       if (call PollObject.resultAvailable ()) {
               //If there is a result...
(16)           call Timer3.startOneShot (time); //send the
                                                    message again!
(17)           call Timer4.stop ();}
(18) }   }
```

Algorithm 4

```
(1) module RCExampleM {...}
(2) implementation {...
(3)    event void Timer1.fired () {
(4)       call NamingService.lookup ("RECOGNITION");}
(5)    event void ResultCallback.result_lookup (uint8_t error,
                  int32_t servId) {//If there is a result...
(6)       call Recognition.setServiceId (* ((int16_t*) servId));
(7)       call Timer2.startOneShot (80000); //Start sending
                                                the message!}
(8)    event void Timer2.fired () {
(9)       call Recognition.isHuman (arrImage, size);}
(10)   event void ResultCallback.result_isHuman (uint8_t error,
                                                    bool result) {
(11)       if (result) {...}
(12)       else {...}
(13)       call Timer2.startOneShot (time); //Send the message
                                                    again!}
(14) }
```

Algorithm 5

Comparing both codes, the main differences are in the events that check/receive the reply. In the Poll Object pattern, those events were timers scheduled by the application itself to check periodically if the reply had arrived (see POExampleM, lines (6) and (14)). Here in the Sync with Server pattern, the events are generated by the middleware infrastructure, which detects the arrival of the reply and notifies the application (5) and (10).

For the sake of clarity, the equivalent code used in a synchronous communication pattern is shown in Algorithm 6.

As the asynchronous ones (`RCExampleM` and `POExampleM`), this code (`SyncExampleC`) just sequentially requests a service that is provided by the Internet (called `RECOGNITION`).

The previously presented codes show that asynchronous patterns split the service request into two different actions: sending the request and checking/receiving the reply. In contrast with that, the synchronous pattern combines those actions in a single line of code. When it sends the lookup request to the Naming Service, it blocks the application

```
(1) module SyncExampleC {}
(2) implementation {
(3)    event void senderThread.run (void∗ arg) {
(4)       call Recongition.setServiceId (call
                              NamingService.lookup (“RECOGNITION”));
(5)       call senderThread.sleep (60000);
(6)       while (TRUE) {
(7)          call Recognition.isHuman (arrImage, size);
(8)          call senderThread.sleep (5000);
(9) } } }
```

Algorithm 6

execution and waits for the result (4). The same occurs when it requests the `RECOGNITION` service (7).

4. Evaluation

This section presents results about some experiments that analyze how those methods affect power consumption in the sensor node.

For all scenarios, we use two IRIS motes [11]: one connected to an MTS400 basic environment sensor board, running the application that constitutes the scenario; the other connected to n MIB520 USB programming board, working as a base station (BS), that is, the sink node. The BS is connected to an Internet host that runs the WISeMid SAGe service or TinyOS SerialForwarder application, which acts as a proxy between WSN and the Internet. Also, two other services run on Internet hosts: the Naming service and the service/application under evaluation.

In order to estimate the power 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 [12], which is responsible for calculating the power 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.

4.1. Aggregation. The first scenario studies the energy-saving offered by the Aggregation service. For that purpose, an Internet service user requests a sensor service that provides the sensed temperature. In the first case, with the Aggregation service off, the Internet service sends 30 requests in a row, with an interval of 100 ms between the messages. The reason for using 30 messages is to make the difference between the energy consumption for both cases more evident without being costly. Sending only one would result in no difference whereas more than thirty would make the measurements slower and difficult to synchronize with the oscilloscope window. The interval of 100 ms between consecutive messages is necessary to assure that the next message will only be sent after the previous one has been completely transmitted. Smaller values have been tried but some messages were still being lost.

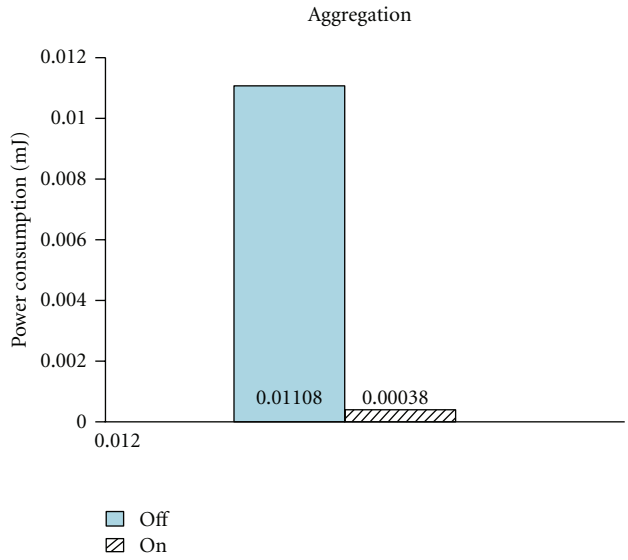

Figure 6: Energy saving by using the WISeMid's aggregation service.

When the Aggregation service is on, only one request is sent by the Internet service user. In this case, the sensor service provider returns one value which corresponds to the aggregation of the last 30 sensed values. The aggregation function is the average.

As Figure 6 shows, the Aggregation service saves 96.57% of energy. That significative energy-saving 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.

4.2. Reply Storage Timeout. SAGe's feature Reply Storage Timeout is evaluated in this scenario. When this feature is on, SAGe stores every WSN Reply message for a given period of time and forwards it to all equivalent Request messages that arrive during this period, avoiding to send Requests that will return the same result (see Section 2.5). To evaluate that, an Internet service user requests the Temperature service 50 consecutive times. We increased the number of requisitions

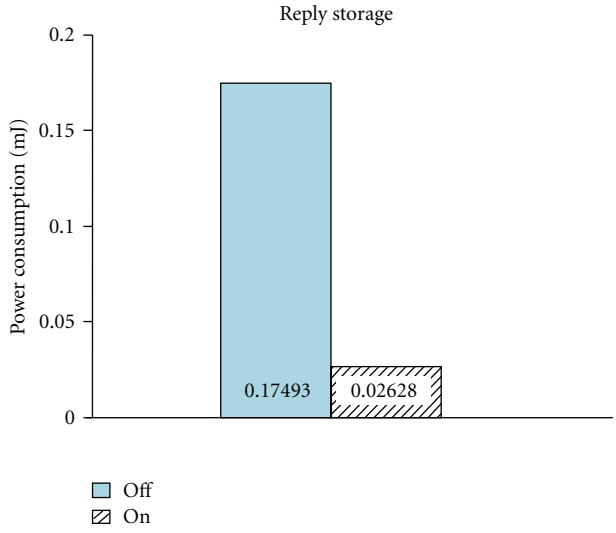

Figure 7: Energy saving by using the SAGe's reply storage feature.

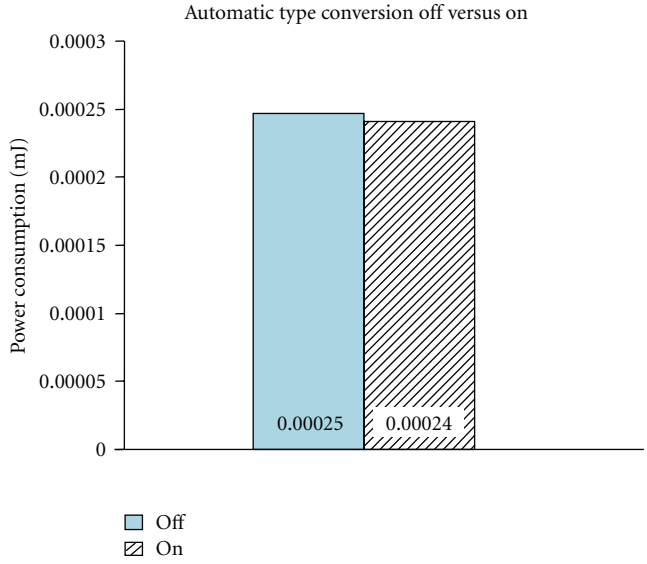

Figure 8: Energy saving by using the SAGe's automatic type conversion.

from 30 to 50 in order to make the difference between the energy consumption for both cases more noticeable. For the same reason, we set the timeout value to 4 s. Also, we decided to make this scenario more realistic by using random intervals between consecutive requisitions. Thus those intervals are now randomly generated following a Uniform distribution with parameters 200 and 400, which allows the reception of 10 to 20 requisitions during the 4 s period a Reply message is stored, depending on the generated values. It is worth mentioning that to compare the power consumption with this feature on and off, the same seed was used for the Uniform distribution in corresponding iterations in both cases. We consider the random intervals between consecutive requisitions to be a more realistic scenario because it represents different clients sending equivalent requisitions during the observed period of time (4 s).

Figure 7 shows that, by avoiding the transmission of unnecessary requests to the WSN, this SAGe feature saves 84.98% of energy in the studied scenario, which is a very significant energy-saving rate.

4.3. Automatic Type Conversion. To evaluate the Automatic Type Conversion performed by SAGe, a simple service is provided by the sensor. This service has an operation that sets an attribute of type `long` (an integer that is represented in 8 bytes) and returns an acknowledgment. An Internet service user 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.

Similarly to the Aggregation scenario, an Internet service user sends 30 requests in a row to the sensor service provider, with an interval of 100 ms between the messages.

The results for this evaluation are illustrated in Figure 8. For the described scenario, the energy-saving gain is 4% when the Automatic Type Conversion is performed by SAGe. Although this is not a very expressive gain, it is still an important gain as every little bit of energy that is saved in a wireless sensor network contributes to preserve and extend its lifetime.

4.4. Invocation Asynchrony Patterns. The Invocation Asynchrony Patterns have to be grouped to be compared. The two patterns that enable one-way operations are compared whereas the two patterns designed for request-response operations are analyzed together.

4.4.1. Fire and Forget versus Sync with Server. The one-way operation scenario is composed of a Logging service implemented as a remote service provider in the Internet (see Figure 9). Sensor services use this service to record log messages, and this action must not influence the execution of the service user. In both cases, the service user sends 30 messages to the Internet with an interval of 100 ms between them. The reason for using 30 messages is to make the difference between the power consumption for both cases more evident without being costly. Sending only one would result in an almost unnoticeable difference whereas more than thirty would make the measurements slower and difficult to synchronize with the oscilloscope window. The interval of 100 ms between consecutive messages is necessary to assure the next message will only be sent after the previous one has been completely transmitted. Smaller values have been tried but messages were still being lost.

Since SAGe acts like a service proxy between both networks, we decided to develop two different implementations of Sync with Server. One synchronizes with the service provider as the pattern states. The other one actually

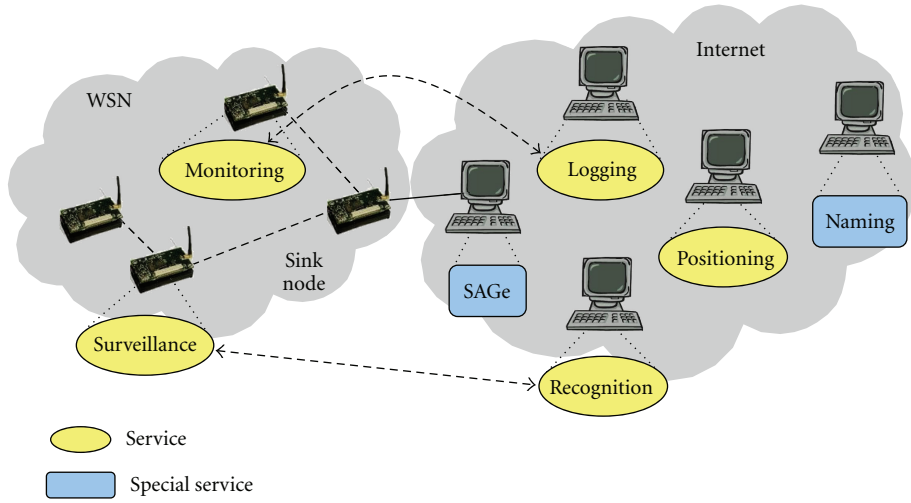

Figure 9: Scenario.

synchronizes with SAGe, which means that SAGe will acknowledge the reception of the invocation instead of the real service provider at the Internet. Figure 10 illustrates the results for both patterns including the two implementations of Sync with Server.

As expected, Fire and Forget is the one that spends less energy, since it ends the process the moment the invocation is sent. On the other hand, the higher power consumption happens when the sensor service user (the client) synchronizes with the Internet service provider (i.e., the server). Comparing these two patterns, the sensor saves 61.27% of energy when Fire and Forget is adopted. That huge difference is due to the tasks performed in SAGe (such as message translation) plus the round-trip delay time from SAGe to the server, when Sync with Server is used.

As the middle ground, when the service user synchronizes with SAGe, it consumes 42.79% less energy while waiting for the acknowledgment from SAGe than if it synchronized with the service. Nevertheless, using Fire and Forget pattern still saves 32.29% of energy when compared to this alternative approach.

4.4.2. Poll Object versus Result Callback versus Synchronous Invocation. The request-response scenario consists of a Monitoring sensor service which captures images when the presence of any moving target is detected but should only report human presence. Therefore, it has to perform an image recognition procedure to avoid reporting the presence of a cat, for example. As this kind of procedure is too computationally demanding and resource consuming, it requests an image recognition service that is provided by an Internet host (see Figure 9).

The first pattern we analyzed was Poll Object. The time interval it waits to check if the answer has arrived affects the sensor service performance. If that interval is very short, this pattern behavior may have to check many times for an answer before it really receives it. On the other hand, if it waits too long, it may waste time waiting for an answer that has already arrived. For that reason, we analyzed how the checking time interval affects the power consumption of a sensor node. Figure 11 presents the results. For the scenario in study, the results show that 50 ms is the best choice for the checking time, as it has the smaller power consumption.

Now comparing the power consumption of a sensor node that adopts the Poll Object pattern, with a checking interval time of 50 ms, to a node that uses synchronous communication pattern (see Figure 12), we can observe that Poll Object pattern saves 43.91% of energy. Comparing Poll Object to the Result Callback pattern, the energy saving is 12.34%, which is still a significant rate. Finally, comparing the synchronous communication to the Result Callback, which has the best results, we can obtain 50.83% of energy saving.

4.5. 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 (see Section 5.1). It may be even less costly in terms of power consumption to request the service outside the WSN. This section evaluates an environment monitoring application that shows this strategy. This application is composed of a Monitoring service that runs on a sensor node and a Positioning service which estimates the sensor node position and is provided by an Internet host. Sensor nodes are mobile and must discover its location/position before reporting measured data (see [13] for examples of real applications with mobile sensor nodes).

This application has two possible implementations: 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 positions are calculated and sent back to the network. In the distributed

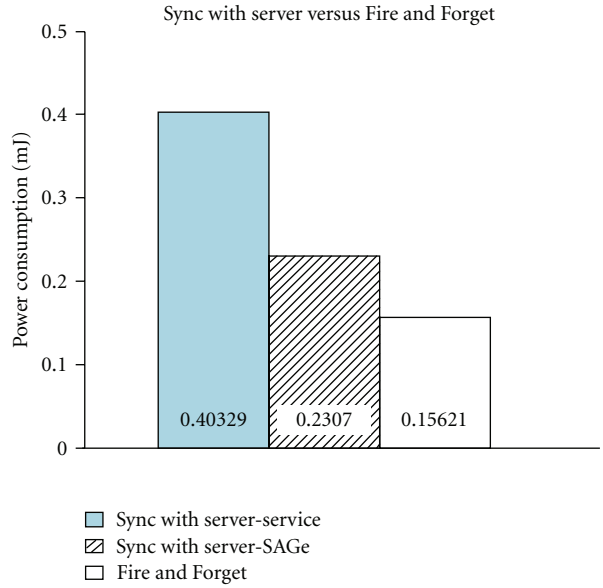

Figure 10: Power consumption of fire and forget and Sync with server methods.

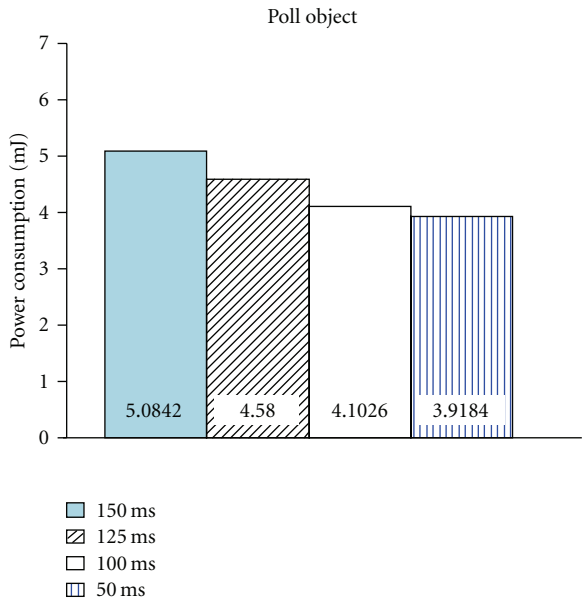

Figure 11: Power consumption of poll object with different checking time intervals.

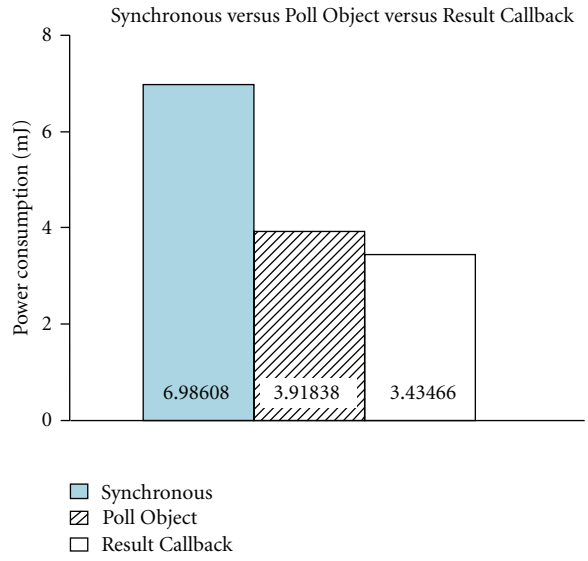

Figure 12: Power consumption of poll object, result callback and synchronous methods.

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 [14, 15].

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 (Positioning service) that is provided by the Internet. It is worth mentioning that synchronous communication is being used, therefore this power consumption includes the time the sensor keeps waiting for the reply (i.e., the computation time of the algorithm plus the round-trip delay time which involves SAGe's tasks). 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. Those values are presented in Figure 13.

For the distributed version, the values shown in Figure 13 represent the estimated power 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 [14] and different types of messages [15]. 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 shows the estimated power 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 that 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.

It is important to emphasize that those are particular results which are specific for this application using this localization approach.

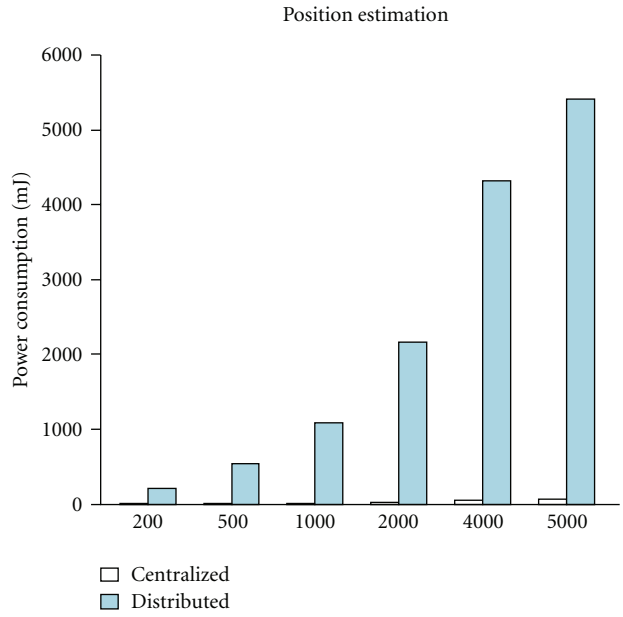

Figure 13: Power consumption of a sensor position estimation method being performed by the WSN or by the Internet.

5. Related Work

This section presents relevant previous works related to the integration of WSNs and the Internet as well as middleware for WSNs.

5.1. Integrating WSNs and the Internet. Some approaches have been proposed to integrate WSNs and the Internet. The simplest one is the gateway-based approach, which may use an application layer gateway, translating query messages from one side (typically the Internet) into messages that can be understood on the other side (usually the WSN) [1, 4, 16, 17], or a Delay Tolerant Networks (DTN) gateway, providing interoperability between and among WSNs, which are considered DTN networks [18, 19].

Overlay-based approaches have been proposed, where some sensor nodes use the TCP/IP protocols or some hosts use WSN protocols [20]. Also, mobile agents have been used to dynamically access the WSN from the Internet [21].

Although directly employing the TCP/IP suite in the WSN would enable its seamless integration with TCP/IP networks, this approach has several problems [22]: the addressing and routing schemes of IP are host-centric and does not fit well with the sensor network paradigm, where the main interest is the data generated by the sensors and the individual sensor is of minor importance and therefore uses data-centric routing and addressing; the header overhead in TCP/IP is very large for small packets, and its size may constitute nearly 90% of each packet when sending a few bytes of sensor data, which is not acceptable as it wastes valuable energy in radio transmission; TCP does not perform well over wireless links networks where packets frequently are dropped because of bit errors; the end-to-end retransmissions used by TCP consume energy at every hop of the retransmission path. Also, sensor nodes memory and computational resources are limited and not able to run a full instance of the TCP/IP protocol stack. For this problem, some works have proposed simplified versions of TCP/IP protocol stack, like [23, 24].

A reflective, service-oriented middleware for WSN is proposed in [25]. The middleware acts as a broker between applications and the WSN, translating application requirements into WSN configuration parameters. It monitors both network and application execution states, performing a network adaptation whenever it is needed. From an external point of view, applications are service requestors and sink nodes are service providers, releasing the descriptions of the WSN services and offering access to these services. From an internal point of view, sinks are the service requestors, and sensor nodes are the service providers. Sensors send the descriptions of their services to sink nodes, which keep a repository of the service descriptors of each type of existing sensor in the network. Although it has some similarities with our proposal, as it is a service-oriented middleware, with service providers and consumers (requestors), it focuses on network adaptation capability. Besides, it assumes that the WSN is a service provider, but not a service consumer.

The implementation of tiny web services directly on sensor nodes is presented in [5], including an XML parser, an HTTP server, and a simplified TCP/IP protocol stack. As the previous approach, it considers the sensor nodes as being only service providers (actually, web service providers), not consumers.

Unlike most of the mentioned works, this paper proposes a solution that focuses on integrating the Internet and WSN at service level instead of integrating protocol stacks and/or mapping logical address. Also, even though some proposals

are service oriented, they only allow requesting services offered by the WSN, as most integration approaches only focuses on accessing the sensor nodes data from the Internet and not the other way around.

Although this is the usual situation, as sensors are typically measured data providers, there are some cases where it is better for the sensor to request a service outside the WSN. There are basically three cases or three types of services that are more suitable to be requested from outside the WSN than being executed by the sensor nodes.

The first type comprises the applications that are too computationally demanding. Applications that require the accurate positioning of sensor nodes usually fit in this category. The simplest way to determine a sensor node location is to equip this node with a global positioning system (GPS); however this is currently a costly solution in terms of energy consumption, hardware cost, and computation capabilities. For that reason, the usual solution consists of equipping only a few nodes with a GPS and using the information about their location as input for some method to estimate the position of the other nodes. The accuracy of those localization methods is usually related to the computational complexity of the algorithm employed. Intuitively, the more complex, the localization algorithm is, the better accuracy can be obtained [15]. Therefore, accurate localization methods are not suitable to run in sensor nodes as they have limited resources. Rather, they can be executed on a central machine with plenty of computational power. Each sensor node gathers the measurements of distances between it and all the neighbors and passes them to the central station where the positions of nodes are calculated. Then, each node may request its computed position instead of calculating it. Environmental sensing applications, such as water quality monitoring, precision agriculture, and indoor air quality monitoring, are good examples since sensing data without knowing the sensor location is meaningless for those applications [26].

The second type of services that can be performed by an Internet host and requested by a sensor node includes those which need to store a huge amount of data (usually in databases). As sensor nodes have a restrict memory capacity, they are not able to store those data. An example of this type can be adapted from the electronic nose proposed in [27]. Defined by the authors as a system capable of recognizing different gases using signal processing and pattern recognition techniques, this electronic nose uses a data warehouse to efficiently store the processed data about the known gases. A single gas sensor composes the nose, and the data warehouse information is used to classify the "smell" sensed by the nose as one of the known gases. This structure may be used, for instance, in a gas leakage detection application of a factory that handles different types of gas (some dangerous and some harmless). In that scenario, a number of gas sensors would be spread throughout the factory forming a WSN, and there would be an Internet node hosting the data warehouse and offering a service for classifying a given "smell" as one of the known gases. When a sensor node detects a "smell," it requests this service to identify the gas, and, if the gas is classified as a dangerous one (i.e., inflammable or harmful to the workers health), it notifies some monitoring application or sounds an alarm.

The third type of services that are more suitable to be requested from outside the WSN contains applications that require a global knowledge of the wireless sensor network. Monitoring applications which fire alarms fit in this category. In the last years, alarm management has been a growing research area on industrial automation as some accidents showed that excessive number of alarms may confuse decision makers. Hence one of the major challenges in this area is alarm rationalization, in which the volume of generated alarms is reduced to an appropriate number so that a human being can handle them [28]. For that purpose, the alarm rationalization needs to have a global vision of the entire monitoring area in order to identify unnecessary alarms and suppress them. Consider the previous scenario with a gas leakage detection application of a factory. Adding the alarm rationalization to this scenario, after detecting a dangerous gas leakage, a sensor node asks an Internet host that performs the rationalization service whether the alarm should be fired.

Alarm rationalization includes identifying, between files and databases containing tens of thousands of daily records, patterns that might indicate unnecessary alarms. Hence, it also fits the other two categories, since it needs to store a big amount of data and has to perform complex computation over those data.

5.2. Middleware for Wireless Sensor Networks. A number of middleware for wireless sensor networks has been proposed in the last years [29–31]. Those solutions may be classified into five categories, namely, distributed database, tuple space, event based, mobile agent, and service oriented.

The distributed database approach treats the WSN as a large "virtual" distributed data base. Existing middleware systems and middleware follow this approach, such as COUGAR [32], TinyDB [33], Global Sensor Networks (GSN) [34], and SINA [35]. These middleware systems perform in-network processing, such as data aggregation, to conserve energy by reducing the amount of communication between sensor nodes. Additionally, Cougar and TinyDB implement a query optimizer that determines energy-efficient query routes, whereas SINA incorporates low-level mechanisms for hierarchical clustering of sensors for efficient data aggregation and protocols that limit the retransmission of similar information from geographically proximate sensor nodes.

TinyLIME [36] follows the tuple space approach, which is based on Linda [37], a shared memory model where the data is represented by elementary data structures called tuples and the memory is a multiset of tuples called a tuple space. In this model, applications add and read/remove data from a common tuple space using "in" and "out" operators. TinyLIME supports application-tunable energy utilization by allowing the user to turn on and off logging of values on the sensors. Logged values are available for efficient aggregation on a single sensor while the basic TinyLIME operations support aggregation across multiple sensors.

For the event-based approach, data acquisition support is focused on the event definition, event register/cancel, event detection and event delivery. Mires [38] is a typical event-based middleware which uses the Publish/Subscribe paradigm, where information providers publish events to the system and information consumers subscribe to events of interest within the system. This approach saves energy by avoiding unnecessary information requests as the sender only sends a message when an event of interest is detected.

The mobile agent approach considers that WSN applications need to be highly flexible and adaptive. Hence, they focus on developing a middleware architecture that enables application modularity, adaptivity, and reparability in wireless sensor networks. In this manner, software updates can be transmitted at the granularity of smaller program module instead of transmitting an entire monolithic program. The key to energy efficiency for middleware that adopts this approach is for the sensor node applications to be as modular as possible, enabling small updates that require little power during transmission. Examples of mobile agent-based middleware are Agilla [39], Impala [40], and SensorWare [41].

Finally, the service-oriented middleware approach aims at providing an abstraction layer between applications and the underlying network infrastructure in order to shield the application developers from the low-level complexities of layers below the application. MILAN [42] and the middleware proposed in [25] follow this approach. MILAN focuses on providing QoS to sensor networks applications. It receives a description of application requirements, monitors network conditions and optimizes sensor and network configurations to conserve energy and maximize application lifetime. The middleware proposed in [25] focuses on the network reconfiguration capability. It adopts aggregation service to save transmission energy and an adaptation policy of decreasing the energy consumption, which may be implemented by two actions: decreasing the data rate and turning off some sensors.

WISeMid may be considered a service-oriented middleware, as it provides an abstraction layer between applications and the underlying network infrastructure, shielding the application developers from the low-level complexities of layers below the application. Unlike all the aforementioned middleware, WISeMid focuses on integrating the Internet and WSNs at service level by providing transparency of access, location and technology. In practice, a service that is offered by a sensor node in a WSN or by a host in the Internet should be accessed in a uniform way irrespective of the location of the client or service. Application developers only need to know the service name to access its operations as WISeMid takes responsibility for hiding the heterogeneity of all network low-level mechanisms. Furthermore, WISeMid is energy aware as it proposes and implements various energy-saving mechanisms, namely, Aggregation service, Reply Storage Timeout, Automatic Type Conversion, the implementation of invocation asynchrony patterns, which have been described and evaluated in this paper.

5.2.1. Middleware Evaluation Comparison. As described in Section 5.1, the middleware proposed in [25] allows requesting data provided by WSN nodes from the Internet, although it is not its focus. The Global Sensor Network (GSN) also allows accessing the sensor data through its virtual sensors.

Besides those, there are further middleware systems that provide certain integration among WSNs and the Internet. Many of those middleware are not developed specifically for WSNs, but they can be used for that purpose. Some examples are listed as follows: HYDRA (networked embedded system middleware for Heterogeneous phYsical Devices in a distRibuted Architecture) [43], MORE (network-centric Middleware for grOup communication and Resource sharing across heterogeneous Embedded systems) [44], ANGEL (Advanced Networked embedded platform as a Gateway to Enhance quality of Life) [45, 46], SOCRADES (Service-Oriented Cross-layer infRA-structure for Distributed smart Embedded Systems) [47] and AMIGO (AMbient IntelliGence for the networked hOme environment) [48]. Those middleware systems focus on enabling communication among different devices (including embedded sensor devices) over the Internet (actually, all-IP networks).

It may seem suitable to compare our middleware to those previous ones; however the different goals as well as the lack of quantitative evaluations make this comparison unattainable. Among the middleware listed above, the only one that performs some energy consumption evaluation is the middleware proposed in [25]. However, its evaluation consists in comparing the network residual energy with and without its adaptation policy being used, as its main purpose is to provide network adaptation capability. The GSN evaluation is about its scalability with respect to the query processing latency and internal processing time, which is not comparable to our focus in this paper. For all the other middleware, no quantitative evaluation has been presented so far. For those reasons, we have not presented comparisons among WISeMid and those middleware in terms of performance or energy consumption.

6. Conclusions

In this paper, we extended WISeMid, a middleware that integrates WSNs and the Internet at service level, by proposing, implementing, and evaluating some mechanisms for saving energy in Wireless Sensor Networks.

The proposed mechanisms include an Aggregation service which aggregates the last n data sensed by a sensor node; a SAGe feature, namely, Reply Storage Timeout, that stores every WSN Reply message for a configurable period of time and forwards it to all equivalent Request messages that arrive during this period, avoiding to send Request messages that will return the same result; an Automatic Type Conversion that aims at removing unnecessary bytes from the sensor messages by converting actual argument types into compatible types that use less bytes to represent the same value; the implementation of invocation asynchrony patterns, which enable the client to resume its work immediately after a remote invocation is sent, preventing the sensor

application from wasting power for being blocked during a service requesting.

The evaluation results lead to the conclusion that using the proposed Aggregation service, a sensor node can save 96.57% of energy, whereas 84.98% less energy is consumed if the SAGe's Reply Storage Timeout feature is enabled. The Automatic Type Conversion saves 4% of energy, and, although it is not an expressive gain, it still contributes to the network lifetime extension, mainly when used along with other strategies. Also, for services that offer only one-way operations, an asynchronous pattern may be adopted. When it is necessary to receive an acknowledgment that the operation has been received by the server, a variation of the Sync with Service pattern may be used, saving 42.79% of energy when it synchronizes with SAGe instead of with the actual server. If that acknowledgment is not essential, the Fire and Forget pattern may be adopted, consuming 61.27% less than the Synch with Server pattern. Finally, for request-response operations, a service may implement the Poll Object pattern and decrease in 43.91% its energy consumption compared to the synchronous communication pattern. Additionally, it may follow the Result Callback pattern and save 50.83% of energy.

In addition to those approaches, we also presented a power consumption evaluation of an application that uses a facility provided by WISeMid: the sensor nodes' capability of being service requestors. Using two versions of an algorithm for estimating sensor nodes position, we compared the power consumption estimation of performing that localization task in network (distributed version) to requesting it outside the WSN (centralized version). The evaluation results have shown that the centralized version saves approximately 98% of energy compared to the distributed version. Although this huge energy saving rate comes from particular characteristics of the chosen application, we believe other applications or even different approaches of this one may also save energy when executed outside the network, probably achieving more modest results.

In terms of future work, further power-saving mechanisms, like a distributed version of SAGe, are now being developed. It is worth mentioning that, besides saving energy by allowing loading balance, this distributed version will also provide fault tolerance to WISeMid since SAGe will no longer be a single point of failure. Furthermore, we are also investigating dynamic adaptation mechanisms, which may be used together with the power-saving mechanisms to automatically reconfigure WISeMid aiming to prolong the WSN lifetime.

Acknowledgment

A. Damaso is supported by CNPq/Brazil.

References

[1] S. Reddy, G. Chen, B. Fulkerson et al., "Share and search: enabling collaboration of citizen scientists," in *Proceedings of the ACM Workshop on Data Sharing and Interoperability on the World-Wide Sensor Web*, pp. 11–16, Cambridge, Mass, USA, April 2007.

[2] Y. Zheng, J. Cao, A. T. S. Chan, and K. C. C. Chan, "Sensors and wireless sensor networks for pervasive computing applications, subsequences," *Journal of Ubiquitous Computing and Intelligence*, vol. 1, pp. 17–34, 2007.

[3] N. Niebert, C. Prehofer, R. Hancock, T. Norp, and J. Nielsen, "Ambient networks—a new concept for mobile networking," Tech. Rep., Wireless World Research Forum, 2004.

[4] P. K. Mohanty, "A framework for interconnecting wireless sensor and IP networks," in *Proceedings of the IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '07)*, pp. 1–3, Athens, Greece, September 2007.

[5] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: design and implementation of interoperable and evolvablesensor networks," in *Proceedings of the ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pp. 253–266, Raleigh, NC, USA, November 2008.

[6] J. Domingues, A. Damaso, and N. Rosa, "WISeMid: middleware for integrating wireless sensor networks and the internet," in *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '10)*, pp. 70–83, Amsterdam, The Netherlands, June 2010.

[7] P. Levis, S. Madden, J. Polastre et al., "TinyOS. An operating system for sensor networks," in *Ambient Intelligenc*, W. Weber, J. Rabaey, and E. Aarts, Eds., pp. 115–148, Springer, Heidelberg, Germany, 2005.

[8] P. Korteweg, A. Marchetti-Spaccamela, L. Stougie, and A. Vitaletti, "Data aggregation in sensor networks: balancing communication and delay costs," in *Proceedings of the International Colloquium on Structural Information and Communication Complexity (SIROCCO '07)*, pp. 139–150, Castiglioncello, Italy, June 2007.

[9] A. Boulis, S. Ganeriwal, and M. B. Srivastava, "Aggregation in sensor networks: an energy-accuracy trade-off," in *Proceedings of the IEEE International Workshop on Sensor Network Protocols and Applications (SNPA '03)*, pp. 128–138, Anchorage, Alaska, USA, May 2003.

[10] M. Volter, M. Kircher, and U. Zdun, *Foundations of Enterprise, Internet, and Real-Time Distributed Object Middleware*, John Wiley & Sons, New York, NY, USA, 2004.

[11] MEMSIC Wireless Module IRIS: IRIS Datasheet, http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html.

[12] E. Tavares, B. Silva, and P. Maciel, "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 '08)*, pp. 291–300, Cape Town, South Africa, November 2008.

[13] K. Römer and F. Mattern, "The design space of wireless sensor networks," *IEEE Wireless Communications*, vol. 11, no. 6, pp. 54–61, 2004.

[14] E. Niewiadomska-Szynkiewicz and M. Marks, "Optimization schemes for wireless sensor network localization," *International Journal of Applied Mathematics and Computer Science*, vol. 19, no. 2, pp. 291–302, 2009.

[15] M. Marks and E. Niewiadomska-Szynkiewicz, "Approach to localization in wireless sensor networks," *Journal of Telecommunications and Information Technology*, vol. 3, pp. 59–67, 2009.

[16] L. Shu, J. Cho, S. Lee, M. Hauswirth, and L. Zhang, "VIP bridge: leading ubiquitous sensor networks to the next generation," *Journal of Internet Technology*, vol. 8, no. 3, pp. 299–311, 2007.

[17] J.-H. Kim, D.-H. Kim, H.-Y. Kwak, and Y.-C. Byun, "Address internetworking between WSNs and internet supporting web services," in *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering (MUE '07)*, pp. 232–237, Seoul, Korea, April 2007.

[18] M. Ho and K. Fall, "Poster: delay tolerant networking for sensor networks," in *Proceedin0gs of the IEEE Conference on Sensor and Ad Hoc Communications and Networks (SECON '04)*, Santa Clara, Calif, USA, October 2004.

[19] M. Loubser, "Delay tolerant networking for sensor networks," Tech. Rep., Swedish Institute of Computer Science, 2006.

[20] H. Dai and R. Han, "Unifying micro sensor networks with the internet via overlay networking," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*, pp. 571–572, Tampa, Fla, USA, November 2004.

[21] J. Bai, C. Zang, T. Wang, and H. Yu, "A mobile agents-based real-time mechanism for wireless sensor network access on the internet," in *Proceedings of the IEEE International Conference on Information Acquisition*, pp. 311–315, Weihai, China, August 2006.

[22] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller, "Connecting wireless sensornets with TCP/IP networks," in *Proceedings of the International Conference on Wired/ Wireless Internet Communications (WWIC '04)*, pp. 143–152, Frankfurt, Germany, February 2004.

[23] A. Dunkels, "Full TCP/IP for 8-bit architectures," in *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys '03)*, pp. 85–98, San Francisco, Calif, USA, May 2003.

[24] M. Durvy, "Poster abstract: making sensor networks IPv6 ready," in *Proceedings of the ACM Conference on Networked Embedded Sensor Systems (SenSys '08)*, Raleigh, NC, USA, November 2008.

[25] F. C. Delicato, P. F. Pires, L. Rust, L. Pirmez, and J. F. De Rezende, "Reflective middleware for wireless sensor networks," in *Proceedings of the 20th Annual ACM Symposium on Applied Computing*, pp. 1155–1159, Santa Fe, NM, USA, March 2005.

[26] N. Patwari, A. O. Hero III, M. Perkins, N. S. Correal, and R. J. O'Dea, "Relative location estimation in wireless sensor networks," *IEEE Transactions on Signal Processing*, vol. 51, no. 8, pp. 2137–2148, 2003.

[27] G. Quispe, *Reconhecimento de padres em sensores integrados*, Ph.D. thesis, Polytechnic School of the University of SoPaulo, 2005.

[28] L. Aguiar, *Descoberta de padres de alarmes redundantes com tcnicas de minerao de dados e redes complexas*, Ph.D. thesis, Federal University of Minas Gerais, 2010.

[29] K. Henricksen and R. Robinson, "A survey of middleware for sensor networks: state-of-the-art and future directions," in *Proceedings of the International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens '06)*, pp. 60–65, Melbourne, Australia, November 2006.

[30] S. Hadim and N. Mohamed, "Middleware for wireless sensor networks: a survey," in *Proceedings of the International Conference on Communication System Software and Middleware (Comsware '06)*, pp. 1–7, Delhi, India, January 2006.

[31] M. M. Wang, J. N. Cao, J. Li, and S. K. Dasi, "Middleware for wireless sensor networks: a survey," *Journal of Computer Science and Technology*, vol. 23, no. 3, pp. 305–326, 2008.

[32] B. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *Proceedings of the International Conference on Mobile Data Management (MDM '01)*, pp. 3–14, Hong Kong, January 2001.

[33] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[34] K. Aberer, M. Hauswirth, and A. Salehi, "A middleware for fast and flexible sensor network deployment," in *Proceedings of the International Conference on Very Large Data Bases (VLBD'06)*, pp. 1199–1202, Seoul, Korea, September 2006.

[35] C. C. Shen, C. Srisathapornphat, and C. Jaikaeo, "Sensor information networking architecture and applications," *IEEE Personal Communications*, vol. 8, no. 4, pp. 52–59, 2001.

[36] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "Mobile data collection in sensor networks: the TinyLime middleware," *Pervasive and Mobile Computing*, vol. 1, no. 4, pp. 446–469, 2005.

[37] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 80–112, 1985.

[38] E. Souto, G. Guimarães, G. Vasconcelos et al., "Mires: a publish/subscribe middleware for sensor networks," *Personal and Ubiquitous Computing*, vol. 10, no. 1, pp. 37–44, 2006.

[39] C. L. Fok, G. C. Roman, and C. Lu, "Mobile agent middleware for sensor networks: an application case study," in *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN '05)*, pp. 382–387, Los Angeles, Calif, USA, April 2005.

[40] T. Liu and M. Martonosi, "Impala: a middleware system for managing autonomic, parallel sensor systems," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 107–118, San Diego, Calif, USA, June 2003.

[41] A. Boulis, C. C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, pp. 187–200, San Francisco, Calif, USA, May 2003.

[42] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo, "Middleware to support sensor network applications," *IEEE Network*, vol. 18, no. 1, pp. 6–14, 2004.

[43] R. Reiners, A. Zimmermann, M. Jentsch, and Y. Zhang, "Automizing home environments and supervising patients at home with the hydra middleware," in *Proceedings of the International Workshop on Context-Aware Software Technology and Applications (CASTA '09)*, pp. 9–12, Amsterdam, The Netherlands, August 2009.

[44] A. Wolff, S. Michaelis, J. Schmutzler, and C. Wietfeld, "Network-centric middleware for service oriented architectures across heterogeneous embedded systems," in *Proceedings of the International IEEE EDOC Conference Workshop (EDOCW'07)*, pp. 105–108, Annapolis, Md, USA, October 2007.

[45] J. Bruynen, M. Nalin, P. Garino, and M. Decandia, "Angel system & platform architecture," in *Proceedings of the 14th IEEE International Conference on Electronics, Circuits and Systems (ICECS '07)*, pp. 625–628, Marrakech, Morocco, December 2007.

[46] E. Alessio, A. Bragagnini, G. Perbellini, and D. Quaglia, "Gateway and middleware design: trusted WSN-TLC network communication and enhanced WSN management," in *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS '07)*, pp. 637–640, Marrakech, Morocco, December 2007.

[47] A. Cannata, M. Gerosa, and M. Taisch, "SOCRADES: a framework for developing intelligent systems in manufacturing," in *Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM '08)*, pp. 1904–1908, Singapore, December 2008.

[48] J. Kalaoja, J. Kantorovitch, M. Karjalainen et al., *Detailed Design of the Amigo Middleware Core Service Modelling for Composability*, IST Amigo Project Deliverable D3.1a, 2005.