Design and Implementation of a UPnP Gateway for Wireless Sensor Networks

Bruno da Silva Campos * ‡, Eduardo F. Nakamura †,
Carlos Mauricio S. Figueiredo † and Joel J.P.C. Rodrigues ‡
*Departament of Computer Science. Federal University of Amazonas, Manaus, AM, Brazil
†Analysis, Research and Technological Innovation Center (FUCAPI), Manaus, AM, Brazil
‡Instituto de Telecomunicações. University of Beira Interior, Covilhã, Portugal

bsc@dcc.ufam.edu.br, {eduardo.nakamura, mauricio.figueiredo}@fucapi.br, joeljr@ieee.org

***Abstract*—Wireless sensor networks (WSNs) are composed of a set of sensing devices to perceive the environment and often perform a common task. Usually the network interaction with other systems or with users is made possible through a gateway. In the other hand, Universal Plug and Play (UPnP) offers a standard interface for users and applications to seamless discover and control devices. Thus, a WSN can be accessed by other systems by using a UPnP interface provided by the gateway, which usually has greater computing power and no energy restriction compared to sensor nodes. In this paper, we present the design and implementation of a UPnP gateway capable of discovering all the services of a WSN, and creating a unique UPnP device to advertise them to interested UPnP control points. Besides, the gateway considers the dynamic behavior of the WSNs and updates the UPnP device when changes occur on the state of the WSN. Experiments in an indoor environment with restricted sensor nodes confirm the feasibility of this solution.**

***Index Terms*—Gateway; Service Discovery Protocol; Universal Plug and Play; Wireless Sensor Networks.**

I. Introduction

Wireless sensor network (WSN) is an ad-hoc network composed of low-power devices called sensor nodes, which perform cooperative tasks in order to do specific activities [1]. Its potential in several application domains has led to the development of several sensor node models, operating on different communication standards and addressing schemes. Hence, designing an application-independent WSNs is not trivial [2]. Therefore, new solutions that provide a uniform and standardized interface have become necessary [3]. Service discovery protocols meet these requirements, since they provide dynamical means for clients and service providers properly discover, configure, and communicate with each other [4].

One of the most used service discovery protocols is Universal Plug and Play (UPnP) [5]. It is designed to enable automatic discovery of devices through standard Internet protocols. The UPnP architecture consists of three main elements: (a) UPnP devices, which have the role of providing services and notifying events occurred during its operation; (b) services, which represent the resources provided by a device; and (c) UPnP control points, which are capable of finding one or more UPnP devices of their interest, control them (using the device services), or subscribe to the device in order to receive event notifications. The interaction between a device and a control point occurs through the following protocols: SSDP (Simple Service Discovery Protocol), which advertise both elements on UPnP; SOAP (Simple Object Access Protocol), which allows a control point to send actions to control a device; and GENA (Generic Event Notification Architecture), that is used to notify subscribed control points about state changes of devices.

UPnP can provide an interface to advertise services of a WSN, making the control of its resources easier through UPnP control points. However, a gateway system in the edge of the WSN is needed to obtain information about the WSN services, and create a UPnP device to advertise them at UPnP. In this paper, we design a UPnP gateway for WSNs. This gateway has the following features: it learns about the services of a WSN automatically; it creates a unique UPnP device to represent the WSN on UPnP; it updates the services provided by the device when the state of the WSN changes; it can operate regardless of data communication protocols. For evatuation purposes, a laboratory testbed has been built in order to show the feasibility of this solution.

The rest of this paper is organized as follows. Section II presents previous solutions to connect to WSNs by using UPnP. Section III describes the gateway architecture and its interaction with WSNs and UPnP control points. Section IV describes an application layer protocol designed to facilitate the exchange of information between the gateway and the WSN. Section V discusses the tests made on the gateway to demonstrate its features. Finally, conclusions and future work are presented in Section VI.

II. Related work

The UPnP stack have been implemented by sensor nodes to provide application-independent WSNs [6], [7]. However, in these cases, sensor nodes must be roubust, while depending of an ip-enabled device to send or receive UPnP messages. Besides, although the creation of UPnP-enabled sensor nodes can make the control of these devices easier, UPnP has several drawbacks, such as implementation complexity and overhead inherent to the used protocols, which makes challenging the use of UPnP directly at the nodes [8]. The network management is also difficult, since each sensor node will be represented by its own UPnP device. So, in large-scale WSNs,

the UPnP will be overloaded with hundreds, or even thousands, of UPnP devices. Because of that, the creation of gateways is an interesting solution to integrate WSNs with UPnP.

Following the same idea, some UPnP gateway architectures have been proposed. The BOSS architecture [9] generates a UPnP device to provide WSN control and management services. However, this UPnP device is not fully UPnP-compliant, because some changes are made on the service description of the sensor nodes. Kim et al. [10] proposed a gateway architecture to create a UPnP device for each active ZigBee sensor node in a WSN, but it is restricted to small-scale WSNs. UPnP has also been used to provide services of a heterogeneous WSN industrial scenarios [11], and to create UPnP devices for each services provided by mutiple WSNs [12].

The gateway herein proposed has some advantages in comparison to these solutions. First, the architecture is flexible to be used in any kind of WSN, independent of its services or the technology used to interconnect the sensor nodes. Second, the gateway also creates a unique and fully compliant UPnP device to represent all services of the WSN, resulting in less overhead. Finally, the gateway provides means for automatic discovery of the WSN services, which has not been discussed on the other proposals.

III. Gateway design

Figure 1 illustrates the interaction among the WSN and users through the proposed gateway. The WSN is the physical entity that offers a group of services for interested users. The UPnP gateway is responsible for discovering WSN services and advertising them at UPnP. In order to do that, the gateway communicates with the WSN through an application layer protocol called USN (UPnP for Sensor Networks), described in Section IV. Once discovered the WSN services and checked that all sensor nodes needed to provide them are activated, the UPnP gateway is able to start the WSN Proxy, which is a UPnP device that offers the WSN services. WSN users are represented by UPnP control points. They are capable of discovering the WSN Proxy, and sending actions or receiving events notification from the WSN. The WSN Proxy is able to communicate directly with the WSN using the USN protocol.

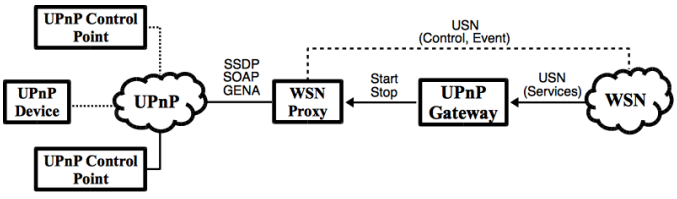


Fig. 1. Basic interaction between clients and the WSN. The lines without arrows denote bidirectional communication between the components.

Figure 2 presents the architecture of the proposed gateway. The core of the system is the Message Manager module. It plays the role of receiving messages about services or data gathered by the WSN. When it is started, its first task is to connect to the WSN. In order to do that, it uses the WSN API module. This module provides an interface to send or receive WSN messages. Hence, it has to support the WSN protocol suite.

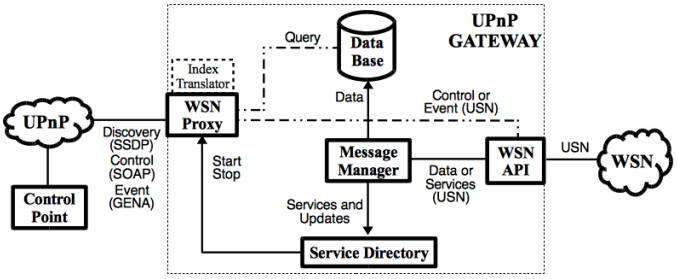


Fig. 2. UPnP gateway architecture.

After connecting to the WSN, the Message Manager needs to discover which sensor nodes are active and what services they provide. In order to accomplish that, it sends a service request message to the WSN. The target sensor nodes reply with a service notification message, informing all services they provide. Service creation messages can also be received to define new WSN services. Furthermore, the Message Manager can also receive messages containing data gathered by the sensor nodes. In this case, the data are stored in a database system, that can be queried by users through specific actions provided by the WSN Proxy.

The data included in the service notification message is extracted by the Message Manager and sent to the Service Directory. This module manages all information related to the services provided by the WSN. For that reason, it needs to know the WSN services, offered by sensor nodes, and be aware of each sensor node state, because its deactivation can disable one or more services. Besides, it starts (or stops) the WSN Proxy execution adapting itself to the changes occurred in the WSN.

The Service Directory is illustrated in Figure 3. When it is started, it receives information about the WSN Proxy (friendly name, model descriptions, among others) from the WSN manager to create its device description. After that, it waits for data incoming from the Message Manager or the WSN manager. In the first case, the Message Manager sends update messages, or data extracted from the service notification and service creation messages, to update sensor nodes or services status in the Service Directory entries, or create new service descriptions. In the second case, the WSN manager interacts with the Service Directory to insert WSN services descriptions, updating the UPnP device information, or adding requirements for WSN services, i.e., which sensor nodes must be activated in order for a WSN service to be provided to clients.

Data sent by the Message Manager or by the WSN manager are handled by the core of the Service Directory. Its tasks include forwarding input data to other components of the Service Directory, namely the Requirement Manager and the Description Creator, maintaining the current status of all sensor nodes in the WSN, and starting/stopping the WSN Proxy when the service requirements are fulfilled or not.

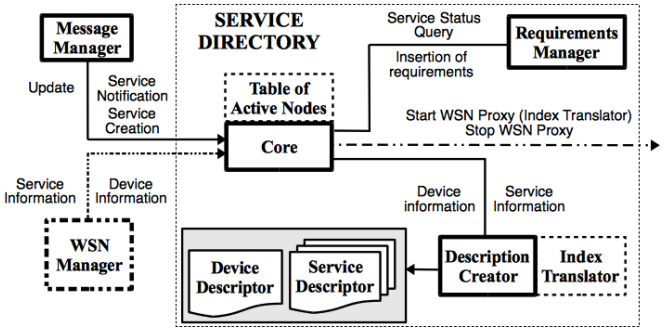


Fig. 3. Internal structure of the Service Directory module.

When the core of the Service Directory receives data about the WSN Proxy or the services of the WSN, it sends them to the Description Creator. This module maintains this information to create or update the device and service description documents of the WSN Proxy. When the gateway learns a service through the WSN (via service creation message), the Description Creator uses the functions of the Index Translator module to convert the indexes sent by the sensor nodes in names of actions or state variables. This function is shown in Figure 4. Service description documents are created as soon as the Description Creator receives all the necessary information to do it. However, the creation of the device description is done only before the initiation of the WSN Proxy. This process will be explained later.

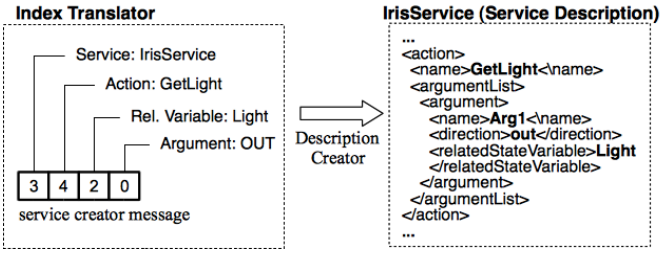
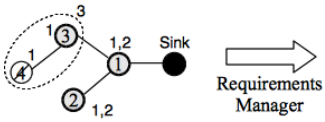


Fig. 4. Illustration of the Description Creator and Index Translator functions, using a illustrative service creator message.

The core of the Service Directory can also receive data related to the service requirements. It needs to know the service requirements to properly create a WSN Proxy that reflects the WSN conditions. The Requirement Manager is the module responsible for handling information about service requirements, and for responding the queries of the core about which services are ready to be provided. It maintains a data structure in order to map a WSN service with its requirements, as it is shown in Figure 5. Besides, it has a flag indicating which services are ready to be provided by the WSN and, hence, by the WSN Proxy.

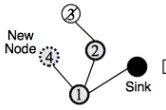
inally, the core of the Service Discovery can also receive update messages. They are used to notify that a specific sensor node is activated in the WSN. When it receives these messages, it updates the Table of Active Nodes, which is used to store the status of all sensor nodes found in the WSN.



Service ID	Requirements	Ready
1	1 or 2 or 3 or 4	Yes
2	1 or 2	Yes
3	3 and 4	No

Fig. 5. Example illustring the function of the Requirement Manager. Node *4* is deactivated.

This table has three fields: (a) the node identifier; (b) a timer that saves the last time the Message Manager has sent an update message; and (c) the current status of the sensor node (activated or deactivated). A register of the table is updated when the Message Manager sends an update message about a sensor node. However, a register is deactivated when its timer expires. These cases are presented in Figure 6. The Message Manager sends an update message about a specific sensor node every time it receives a message from this node.



Timer expiration: 15 min

Node ID	1	2	3	4
Timer	00:03:27	00:08:46	00:15:07	00:00:00
Activated	Yes	Yes	No	Yes

Fig. 6. Example illustring the function of the table of active nodes. Node *3* is deactivated and node *4* has been recently activated.

Whenever the Table of Active Nodes is modified, some actions are taken to start or stop the WSN Proxy. If a sensor node is activated, the core notifies the Requirements Manager about the activation. Then, the Requirements Manager returns the index of all services that are ready to be provided. After that, the core sends these indexes to the Description Creator, that creates a new device description containing a reference for these services, and then returns a copy of the Index Translator. Finally, the core starts the WSN Proxy, transferring the copy of the Index Translator to it (the reason is shown later). When the sensor node is deactivated and one or more services become unavailable, the core does a similar process, but a new device description is created, without the reference of the unavailable services. Besides, if a node is deactivated and all services become unavailable, the WSN Proxy is not started.

After the startup of the WSN Proxy, control points are able to control or receive events notification from the WSN. Figure 7 illustrates how this happens. When a control message is sent by the control point, the WSN Proxy *i*) extracts the action name, service type and input arguments from the message, *ii*) retrieves the action name and service type indexes through the Index Translator, *iii*) encapsulates them (together with any input arguments) in a USN control message, and *iv*) sends it to the WSN. If the action has a result, the action information is queued, and a timer is started. So, if the WSN does not reply before the timer expires, the action information is dequeued, and a SOAP message indicating an error is sent to the control point. If the WSN replies before the timeout, the action information is also dequeued, and a SOAP message containing the returned values is sent to the control point. Control points can also send a database query, through a

specific action, in order to retrieve the data gathered by the WSN. In these cases, the WSN Proxy queries the database with the input argument of the action, stores the result in a text format (such as CSV), and returns it to the control point.

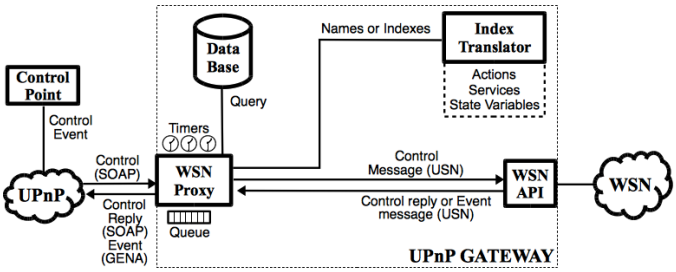


Fig. 7. Interaction between the UPnP Control Point and the WSN Proxy.

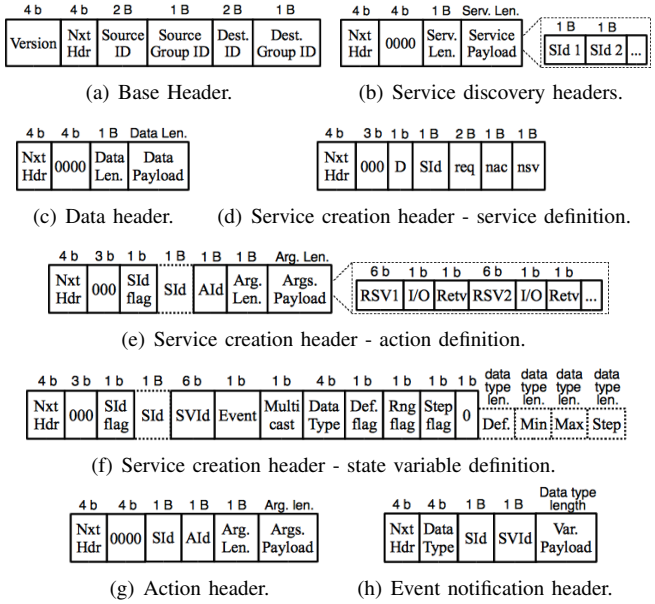
When events occur in the WSN, the following operations are executed by the WSN and WSN Proxy to notify interested control points. First, the WSN sends an USN event message to the WSN Proxy notifying the event. Next, the WSN Proxy extracts the service and state variable indexes, and obtains their corresponding names from the Index Translator. With these names, the WSN Proxy is able to recover the state variable, define a new value, and send a GENA message notifying the event to the control point.

IV. USN PROTOCOL

Several messages are exchanged between the WSN and UPnP gateway components in order to discover services, receive data, control, and notify events of the WSN. As the WSN is not able to process the UPnP protocol stack, a protocol is necessary for both elements to send commands or data properly for each other. For this purpose, the USN (UPnP for Sensor Networks) protocol has been defined. Its design is based on stacked headers (the same technique used in IPv6 protocol), resulting in less overhead with transmission and processing. The basic USN header has the fixed size of 7 bytes, and it can be extended through optional headers, which should be placed after the basic header. Besides, the USN packet is processed only by the recipient of the packet. Figure 8 depicts all headers of the USN protocol.

The basic USN header (Figure 8(a)) stores information about the source and destination of the packet. Besides, it updates the state of the source node in the Table of Active Nodes. The version field specifies the version of the USN protocol. The next header field identifies the following extension header. The source and destination ID fields address the packet for a specific sensor node, or all nodes through a broadcast ID. In the same way, the source and destination group ID fields address a packet for a group of nodes, or all groups through a broadcast ID. In this work, we considered that a group of sensor nodes can be formed when they have the same services.

The service discovery extension header (Figure 8(b)) defines means for the WSN to advertise its services to the gateway, and for the gateway request services from the WSN. The service payload field can contain zero, one, or more service identifiers, i.e., indexes that represent service descriptions of



(a) Base Header. (b) Service discovery headers.

(c) Data header. (d) Service creation header - service definition.

(e) Service creation header - action definition.

(f) Service creation header - state variable definition.

(g) Action header. (h) Event notification header.

Fig. 8. Headers of the USN protocol.

a sensor node. The service length carries the total of service identifiers stored in the service payload. When the gateway requests services, it sends a message with the services it wants (or 0 for all services) in the service discovery header. The destination replies with its services using the same header.

The data extension header (Figure 8(c)) is used by the WSN to send gathered data to the UPnP gateway. Therefore, these data are carried in the data payload field, whereas the data length carries the payload (in bytes).

The service creation header extensions are used by the WSN to automatically create a service description in the UPnP gateway. Three headers are necessary: service, action, and state definition headers. The service definition header (Figure 8(d)) carries basic information about the service, such as the service identifier (SId field), number of actions (nac), number of state variables (nsv), total of sensor nodes that must be activated to the service become available (req), and the dependencies among the nodes (D flag), i.e., whether the service depends of any node to be provided or if it depends on a group. Therefore, it should be sent to the UPnP gateway before the action and state headers.

The action definition header (Figure 8(e)) defines an action of a service. Thus, if a service has n actions, then n action definition headers should be used. It does not need to follow the service definition header. However, when this happens, the service identifier must be carried in the SId field, and the SId flag must be set. The action header carries an identifier (AId field), and a list of arguments, where each one is composed by a related state variable (RSV), a direction flag (in or out), and a return value flag (retv). The total of arguments of the action is specified in the argument length field.

The state definition header (Figure 8(f)) specifies one state variable of a service. This header should be used for each state variable in the service, and does not need to follow the service definition service (but the SId field must be used). The state

variable header has an identifier (SVId), flags to indicate if the state variable send events and if the WSN Proxy should send events using multicast, an identifier of its data type, flags that indicates whether it has a default, a range, and a step values, and their respective values (def, min, max, and step fields).

The action header (Figure 8(g)) carries data to act on the WSN, i.e., send actions to the network and receive corresponding results. Therefore, this header carries the identifiers of the action (AId), the service that owns it (SId), and a list of input or output argument values (in the case of a control message or control reply message, respectively), where its size (in bytes) is specified in the arguments length field.

Finally, the event notification header (Figure 8(h)) allows the WSN to notify events to the WSN Proxy. It comprises the data type of the state variable, as well as its identifier (SVId), an identifier of the service where it has been declared (SId), and a payload to carry its new value (var. payload).

V. Experimental Results

In order to validate the proposed architecture, a laboratory testbed has been built. The WSN side is comprised of five Iris and four Micaz sensor nodes running the TinyOS operating system [13]. The Iris nodes have used a MTS300 sensor board to gather temperature, whereas the Micaz nodes have used a MDA100 sensor board to gather light readings. These readings have been stored in a MySQL database system located in the UPnP gateway. Iris nodes have also been configured to provide the following functions: emit a beep during a determined interval; return the current temperature and battery voltage measurements; return the LQI of a received message; set their leds and their radio wake up interval; and change their data collection interval. In the Micaz nodes, the following functions have been configured: return the current light and battery voltage measurements; return the RSSI of a received message; and set their leds, their radio wake up interval, and their data collection interval. Furthermore, both nodes send event messages when the state of the leds changes. They have used the active message communication model.

The UPnP gateway has been tested in a MacBook with 2.4 GHz processor and 2 GB RAM memory, running the Ubuntu 10.04 operating system through the virtual machine VirtualBox. To allow the gateway communication with the WSN, an Iris node has been programmed with the base station application (provided by TinyOS), and connected to a MIB520 programming board. The UPnP gateway has been implemented using the Java programming language. Thus, the CyberLink API [14] for Java has been chosen to enable it with UPnP functionalities. To show that the WSN Proxy generated by the gateway fulfill the UPnP standards, a generic UPnP control point, available in the CyberLink API, has been used to interact with it. This control point has been tested in a MacBook Pro with 2.5GHz processor and 4 GB RAM memory, running the Mac OS X 10.6.5 operating system.

In order to provide the Iris and Micaz commands, two service descriptions have been used: *IrisService* and *MicazService*. The former has been added manually during the Service Directory startup, whereas the latter has been generated through the service creation message sent by one Micaz node. Using the USN protocol, it has needed 91 bytes to create a service description containing 6 actions and 6 state variables (6 bytes of service definition; 33 bytes of action definition; 24 bytes of state variable definition; and 28 bytes of base header, since four messages have been sent to create the service description). Furthermore, an action has been defined to enable the control point to send queries to the data base. It has been exposed by a third service description, called WSNService.

Figure 9 shows the interaction between the control point and the WSN Proxy generated by the UPnP gateway. Upon the initiation of the WSN, the gateway has learned about the WSN services, and discovered the activated sensor nodes. Then, it has started the WSN Proxy (Figure 9(a)). After that, the control point has been able to discover the WSN Proxy and learn about its services, enabling it control and subscribe to receive event notifications from the WSN Proxy. Figure 9(b) presents the control point sending an action to turn on all leds of the sensor node identified by 3. Figure 9(c) shows it receiving a notification about the state of the leds, after the previous operation. Finally, Figure 9(d) presents the average temperature and light gathered by the nodes, which has been obtained through a query sent to the database of the gateway.

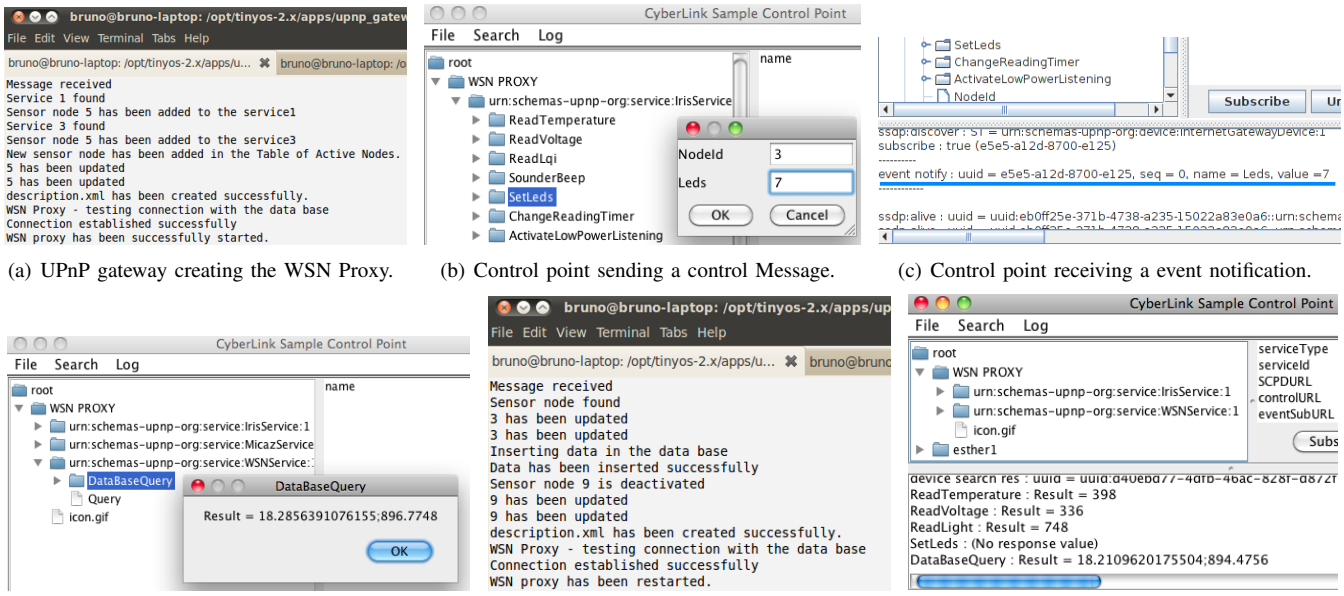
When sensor nodes are deactivated and, hence, some services become unavailable in the WSN, the UPnP gateway needs to update the WSN Proxy. Figure 9(e) presents the UPnP gateway dealing with this case. In this testbed, it has been considered that all registers of the Table of Active Nodes are deactivated when the timer of $3\times$ *data collection interval* expires. Thus, when all Micaz nodes have been turned off, their respective timers in the Table of Active Nodes have expired. After that, a new device description has been created without the references to the MicazServices service description. Finally, the WSN Proxy has been restarted. Figure 9(f) shows the WSN Proxy after these operations.

VI. Conclusions and Future Work

Service discovery functionalities are important for WSN since they provide a uniform communication interface for clients (users or applications) discover and control the WSN resources. This paper presents a contribution for this research area, through the design of a UPnP gateway as a solution to the integration of WSNs with a service-based environment.

The experimental results show that the proposed architecture is flexible, enabling the gateway to obtain services information from heterogeneous WSNs, independent of the communication protocols used by them to transfer sensor data. This flexibility allows us to design WSNs with the technologies that best fit the requirements of the application, increasing their potential. The current design of the gateway architecture does not consider its interaction with more than one WSN. However, this feature can be possible through the addition of a new field in the USN basic protocol to identify the WSNs.

The expiration time of a register in the Table of Active Nodes is an important parameter to measure the reliability of



(a) UPnP gateway creating the WSN Proxy. (b) Control point sending a control Message. (c) Control point receiving a event notification.

(d) Control point receiving the result of a query. (e) UPnP gateway updating the WSN Proxy. (f) Control point after the update.

Fig. 9. Pictures showing the interaction between the generic UPnP control point and the WSN proxy generated by the UPnP gateway.

the UPnP gateway, since it defines the time that the gateway takes to detect the deactivation of some node. Thus, it should be chosen carrefully. Calculate the measured round trip time (MRTT) [15] in the Message Manager can return good values. In heterogeneous WSNs, it is desirable that each register of the table has its own expiration time to improve the reliability.

Further work includes the realization of tests to measure the scalability of the proposed solution, since it can be a concern in large WSNs. Besides, it is necessary evaluate if the quantity of services that a UPnP device provides impacts on the quality of service offered by the gateway. If so, maybe it should be better to create UPnP devices for each services provided by the WSN, as proposed by Isomura *et. al.* [12]. Finally, header compressions or cross-layer optimizations could be applied in the USN protocol to reduce the size of the USN headers.

ACKNOWLEDGMENTS

This work has been supported by the Amazonas Research Foundation (FAPEAM), under the process of reference PIPT 1509/08, by the National Council for Scientific and Technological Development (CNPq), under the processes 47.4194/2007-8, 55.4087/2006-5, and 55.4071/2006-1, by the *Instituto de Telecomunicações*, Next Generation Networks and Applications Group (NetGNA), Portugal, and by the Euro-NF Network of Excellence from the Seventh Framework Programme of the EU, in the framework of the Specific Joint Research Project PADU.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, pp. 393–422, 2002.

[2] F. C. Delicato, P. F. Pires, L. Pirmez, and L. F. R. d. C. Carmo, "A service approach for architecting application independent wireless sensor networks," *Cluster Computing*, vol. 8, pp. 211–221, 2005.

[3] R. Marin-Perianu, H. Scholten, and P. Havinga, "Prototyping service discovery and usage in wireless sensor networks," in *Local Computer Networks. LCN 2007. 32nd IEEE Conference on*, October 2007, pp. 841 –850.

[4] F. Zhu, M. W. Mutka, and L. M. Ni, "Service discovery in pervasive computing environments," *IEEE Pervasive Computing*, vol. 4, pp. 81–90, 2005.

[5] UPnP, "UPnP forum." 2011. [Online]. Available: http://www.upnp.org

[6] Y. Gsottberger, X. Shi, G. Stromberg, W. Weber, T. Sturm, H. Linde, E. Naroska, and P. Schramm, "Sindrion: a prototype system for low-power wireless control networks," in *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference on*, October 2004, pp. 513–515.

[7] D. Barisic, G. Stromberg, and M. Beigl, "Advanced middleware support on wireless sensor nodes," *2nd International Workshop on Design and Integration Principles for Smart Objects (DIPSO2008)*, 2008.

[8] J.-P. Vasseur and A. Dunkels, *Interconnecting Smart Objects with IP - The Next Internet*. Morgan Kaufmann, 2010.

[9] H. Song, D. Kim, K. Lee, and J. Sung, "UPnP-based sensor network management architecture," *In Second International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2005)*, 2005.

[10] S. H. Kim, J. S. Kang, K. K. Lee, H. S. Park, S. H. Baeg, and J. H. Park, "A UPnP-ZigBee software bridge," *International Conference on Computational Science and Its Applications*, vol. 4705, pp. 346–359, 2007.

[11] M. Marin-Perianu, N. Meratnia, P. Havinga, L. de Souza, J. Muller, P. Spiess, S. Haller, T. Riedel, C. Decker, and G. Stromberg, "Decentralized enterprise systems: a multiplatform wireless sensor network approach," *Wireless Communications, IEEE*, vol. 14, no. 6, pp. 57–66, December 2007.

[12] M. Isomura, T. Riedel, C. Decker, M. Beigl, and H. Horiuchi, "Sharing sensor networks," in *Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on*, July 2006, pp. 61 – 61.

[13] TinyOS, "Tinyos operating system." 2010. [Online]. Available: http://www.tinyos.net/

[14] S. Konno, "Cyberlink for java." 2010. [Online]. Available: http://www.cybergarage.org/twiki/bin/view/Main/CyberLinkForJava

[15] K. il Hwang, J. In, N. Park, and D. seop Eom, "A design and implementation of wireless sensor gateway for efficient querying and managing through world wide web," *Consumer Electronics, IEEE Transactions on*, vol. 49, no. 4, pp. 1090 – 1097, November 2003.