Minimalist Object Oriented Service Discovery Protocol for Wireless Sensor Networks

D. Villa, F. J. Villanueva, F.Moya, F. Rincón, J. Barba, and J. C. López

Dept. of Technology and Information Systems
University of Castilla-La Mancha
School of Computer Science. 13071 - Ciudad Real. Spain
{David.Villa, FelixJesus.Villanueva, Francisco.Moya, Fernando.Rincon, Jesus.Barba, JuanCarlos.Lopez}@uclm.es

Abstract. This paper presents a new Service Discovery Protocol (SDP) suitable for Wireless Sensor Networks (WSN). The restrictions that are imposed by ultra low-cost sensor and actuators devices (basic components of a WSN) are taken into account to reach a minimal footprint solution.
The WSN communication model we use is based on the `picoObject` approach [1] which implements a lightweight middleware for WSN on top of standard object oriented middlewares using a small set of interfaces. The proposed SDP uses also this set, so it supposes the minimal overhead for devices and communication protocols, allowing, at the same time, the deployment of a valuable set of services.[1]

1 Introduction

Wireless Sensor Networks (WSNs) are called to be a key component in any pervasive environment, supporting the interaction (monitoring and driving) with the physical world. A WSN is composed of low-cost nodes which contain three types of elements: a sensor or an actuator, a generic microcontroller and a network interface. Sensors and actuators are oriented either to monitorize a physic magnitude (e.g temperature, humidity, smoke, etc.) or to modify the state of an element which drives such a physical magnitude (e.g a valve). The microcontroller basically adapts raw data and provides communication facilities for applications. At last, the network interface offers wireless network connectivity.

Flexibility and quickly deployment (due mainly to their wireless interface) are the characteristic that make WSNs to become a good solution for multiple applications such medical [4] or meteorology [5] applications, habitat monitoring [6], etc. In general, we can envision a pervasive environment plenty of heterogeneous WSN nodes offering different services, from the most basic (supported by individual nodes or the whole network) to the most complex (ambient intelligent services resident in the environment).

[1] This research is partly supported by FEDER and the Spanish Government (under grant TIN2005-08719) and by FEDER and the Regional Government of Castilla-La Mancha (under grants PBC-05-0009-1 and PBI-05-0049)

However, the flexibility in the deployment of WSN (avoiding wiring) has not found its counterpart when developing software for such a type of networks. We believe that a real deployment of a WSN has to minimize also the configuration requirements of the application that take advantage of the services supported by every WSN node. With the service discovery protocol (SDP) described in this paper, a WSN node has the capacity to announce its services and offer the possibility to use them without any previous configuration procedure.

The proposed SDP: a) Allows very low-cost nodes to be deployed in an easy and incremental way (following a *Place & Play* philosophy). b) Allows applications to discover and use the services offered along a WSN (such property is really desirable in mobile applications). c) Is designed for heterogeneous WSNs where different nodes have different functionalities and even are implemented in different technologies.

The SDP described in this paper is based on our previous work called `picoObject` [1]. As we report in that reference, this approach allows a very high degree of interoperability with standard distributed object oriented middlewares, and provides also the capability to view and to use the WSN nodes as conventional distributed software objects without any intermediate device. The strong footprint limitations determine the design of a `picoObject`, as well as the design of our SDP (as we will show in the next sections).

The SDP prototype is based on ICE [17] (Internet Communication Engine), a high quality distributed object framework developed by ZeroC, Inc. built upon the experience of CORBA but free of legacy or bureaucracy constraints.

The rest of this paper is organized as follows. Section 2 explains some previous works on SDPs. In section 3 the `picoObject` approach is briefly summarized. Section 4 is devoted to explain our SDP in detail. In section 5 the prototype we have used to validate our proposal is briefly described. Finally we draw some conclusions and outline some future work.

2 Related work

In the last years, several SDPs have been designed with the aim of automatizing the service discovery and minimizing the configuration procedures required to integrate a service in any networking environment.

Broadly used currently, some SDPs like UPnP [8], JINI lookup service [16], Bluetooth SDP [10] or SLP [9] are considered as the de facto standards. The evolution of fields like ambient intelligent, pervasive computing, or ubiquitous computing has made it possible the development of an important amount of services that use a variety of heterogeneous technologies and that need to interoperate. This growth of services inherently implies complex configuration procedures for integration with other networks services. Consequently, serious efforts have to be made in order to simplify such configuration procedures and to make it possible to support mobile services and service interoperability.

However, the current SDPs are not suitable for WSNs due to the serious footprint restrictions the WSN nodes impose. Such restrictions have to do with power supply, memory limitations, processing capacity, etc., parameters that have not been taken into account in the design of current SDPs. For example, due to footprint limitations, neither an XML parser (like UPnP requires) nor a Java Virtual Machine (needed by the JINI lookup service) could be implemented in a WSN node. Even lightweight protocols oriented to mobile devices like Bluetooth SDP or PDP [13] do not assume such constraints in their design.

Recent works have proposed SDPs for new technologies like mobile ad-hoc networks [12] and [11]. In these highly dynamic environments, in which services are registered in a directory (in a similar way to yellow pages), the directory-based structures cannot be deployed due to the lack of a fixed infrastructure. This has been the problem usually addressed, but, once again, the minimal footprint requeriments of WSN nodes have not been taken into consideration.

On the other hand, current platforms oriented to support WSN (good surveys can be found in [15] and [14]) are working prototypes in which the nodes will have to be reduced in cost (therefore probably in resources) for a eventual massive introduction in the market.

In [7] a resource discovery protocol (called DRD) specially designed for WSN is described. In DRD each node sends a binary XML description to another node that has been selected as the cluster head (CH) (this node assumes the representation of all the nodes under its range) which responds to any possible query (in SQL) in place of its cluster sensors. The CH is selected between all the nodes depending on their remaining energy. Thus it is necessary to give all the nodes the capacity of being a CH. This means that all nodes need SQLlite database, libxml2 and a binary XML parser to implement the CH functionality. Our approach, as we will describe in section 4, provides a way to incrementally add functionality to the nodes, so ultra low-cost sensor nodes can be easily integrated in a first step and, then, according to its capacity, acquire new functionality. It is necessary to clarify that when we are talking about wireless sensor nodes we are thinking on a minimal footprint device, even more limited than current prototype platforms like MICA, MicaZ, RockWell WINS, etc.

Finally, in [3] an homogeneous sensor network (all the nodes have the same functionality) resource discovery protocol is proposed, centering in the optimization of the flooding process by taking advantage of historical queries [7]. Our work supposes that a WSN is formed by heterogeneous nodes implementing different services that do not need to be considered in an homogeneous way (managed by a simple table).

In general, we observe that previous works have not faced the design of SDPs in such a way that: a) they turn out to be suitable for heterogeneous WSN, taking into account the footprint requeriments of small devices, and, 2) they support the use of node services by client applications without the need of a configuration procedure. Therefore, we will focus on these issues.

3 picoObjects

Our SDP has been designed to give support to WSN based on picoObjects, although it is perfectly applicable (without any change) to more powerful devices or even to WSN based on other approaches (including, for example, some widely used devices such as the MICA Motes).

The picoObjects are implemented as message matching automatons. From a textual description (that includes the object interface description), the pico-Object compiler can generate these automatons in several programming languages and for several platforms.

This approach allows the picoObjects to be embedded either into the smallest microcontroller in the market, into the tiniest embedded Java virtual machine, or even in a low-end FPGA. For a deep description of the picoObject approach, please refer to [1]. A picoObject implementation example can be found in our webpage [18].

4 Abstract Service Discovery Framework

We have defined an ultra lightweight service discovery protocol, called ASDF (*Abstract Service Discovery Framework*), which, using the object oriented paradigm, provides several valuable features such as: a) An easy way for device announcement. b) Extensibility and scalability. c) Legacy SDP interaction. d) Seamless integration with standard middlewares. e) Auto-configuration for devices (in order to get a *place & play* behavior).

The ASDF is designed keeping in mind minimal footprint devices. For example, the protocol allows the nodes to announce themselves to the network using simple, but completely middleware compliant, messages. In spite of this, the protocol is very scalable and can perfectly be applied to more powerful devices.

4.1 Event Channels

Our protocol uses extensively the middleware standard *event* service. This makes it possible to easily decouple all involved elements. The event channel is a direct implementation of the *observer* [2] design pattern (also known as *publish-subscribe*).

The IceStorm (the ZeroC ICE *event channel* service) is able to employ several transport protocols at same time (at least TCP, SSL, UDP and multicast UDP) in a transparent way for objects and even over the same channel. Each publisher or subscriber can even choose the protocol to use individually.

However, it is not convenient to connect too many nodes to the same event channel due to scalability reasons. Therefore, several event channels (*topics* in ICE parlance) are used. Event channels have minimal resource cost and they can be interconnected by means of “links” to propagate events to each other. These links have some parameters that allow to establish limits or priorities to the event propagation.

Event channel *federation* is another technique to group some nodes (their corresponding event channels) together according to different criteria (functionality, location, class of service...) in the same logic channel, but keeping the ability to propagate certain events to other channels.

4.2 Place & Play environment

Node deployment is a key issue for sensor networks. It is very convenient that nodes can configure themselves in an autonomous way. When an actor (an actor is a node/device that can expose its functionality by means of an object interface) is connected o returns from a sleep state, the node sends an announcement message (`adv()`) to a specific event channel (called *ASD.announce*). Optionally, these announcements can also be sent periodically. The `adv()` member function is part of the `iListener` interface. Because of this, all the applications or actors that are interested in announcing their services, must implement the aforementioned interface. The description of this interface is as follows:

```
module ASD  {
  interface iListener {
    idempotent void adv(Object* prx, iProperties* prop);
  };
};
```

The argument `prx` is a proxy to contact the object that sends the event. The argument `prop` is an object that serves to access the node properties (see 4.3). The next listing exposes the content of an `adv()` message:

```
Magic Number: 'I','c','e','P'
    Protocol: 1,0 - Encoding: 1,0
    Message Type: Request (0)
    Compression Status: Uncompressed (0)
    Message Size: 54,  Request Message Body
        Request Identifier: 0
        Object Identity Name: publish
        Object Identity Content: asdf
        Operation Name: adv - Ice::OperationMode: normal (0)
        Input Parameters Size: 16
        Input Parameters Encoding: 1,0 - Encapsulated parameters (10 bytes)
```

Sometimes, the `adv()` message arguments are fully static. In these cases, since the total message size is about 80 bytes, these arguments can be stored in the device ROM.

The clients and services interested in the potential announcements that may occur must subscribe to the event channel *ASD.announce*. When a subscriber receives an `adv()` event, it gets the object proxy of the announced actor and uses the introspection mechanisms to interrogate the actor. The subscriber can also list and request the actor properties by means of the argument `prop`.

Although this announcement procedure has a high abstraction level, it can be implemented on very simple devices with an identical behaviour respect to a conventional “object”.

4.3 Properties

As mentioned before, the parameter `prop` in the `adv()` message is an object proxy for a “property server”. The property server allows the clients to access the actor properties. There are several alternatives:

– The argument `prop` can be a null proxy when it is not necessary or there is not a property server for the actor.
– The proxy `prop` can point to a remote object in a different localization. This allows to implements corrective property servers for many actors whose properties are stored out of the actor, even in a big database. A single servant can dispatch many objects using a “default servant” strategy.
– If the device has enough computing resources, the property server can be implemented in the own device. In this case, both `adv()` arguments, `prx` and `prop`, point to the same object.

The property servers implement the `iProperties` interface:

```
module ASD  {
  interface iProperties {
    Ice::ByteSeq propget(string key);
    void propset(string key, Ice::ByteSeq value);
    Ice::StringSeq proplist(void);
  };
};
```

The properties are specified by means of a string key. The property value is a byte sequence and thereby it can store strings, configuration files, binary drivers, images, maps, Java applets, etc

In any case, the actor properties are considered optional -not required- information. This information is useful for administration, configuration and monitoring tools but it doesn’t affect the system basic functionality. The system services never depend on property values or their availability.

4.4 Basic interface for actors

All actors (sensors or actuators) implement a very simple interface to expose their state value. The sensor state is the measured value of the physical magnitude. There are different interfaces that depends on the type of data they manage. Some of them are shown next:

```
  module iBool {
    interface W { void set(bool v); };
    interface R { nonmutating bool get(); };
  };

  module iByte {
    interface W { void set(byte v); };
    interface R { nonmutating byte get(); };
  };
  ...
```

4.5 Interaction model for actors

Depending on the application interacts with actors, there are four basic types of actor behaviors: **Passive**) To get the state value of a passive sensor, the client needs to invoke explicitly the actor's `get()` method and then will receive the reply in a synchronous way. **Active**) The active actor is able to send a `set()` message in a pre-programmed way to another object (usually an event channel). That message indicates the current state of the actor. **Proactive**) It's also an active sensor but it sends the `set()` event when a change occurs in its state. **Reactive**) A reactive sensor is an active sensor that sends `set()` events only if a client invokes its standard `ice_ping()` method. The `ice_ping()` standard functionality has been extended so when this method is invoked, the actor, besides the conventional `ice_ping()` behaviour, sends an event to the pre-defined event channel to publish its state.

Therefore, when we talk about active actors (or active sensors), we refer to both, reactive and proactive ones. All active objects implement the interface `iActive` that is shown below:

```
module ASD  {
  interface iActive { idempotent void topic(Object* prx); };
};
```

The passive actors requires a two-way communication model while the active ones could use a one-way communication model.

Using the `topic()` method, an specialized service can instruct the actor about the remote object (event channel) where the actor must send its events.

4.6 Actor set-up

The active sensors need an event channel to send their state updates. When an actor announces itself, a "channel monitor" service does the following tasks (figure 1):

1) Using the middleware introspection features, it asserts that the new actor is actually an active actor (it implements the `iActive` interface). 2) It creates an event channel using the object identity as the channel name. If that event channel already exists (it has been created before) then no further actions are needed and the process finishes. 3) After creating the corresponding event channel, the monitor invokes the actor's `topic()` member function with the proxy for the new event channel as the argument.

This process is designed keeping in mind that actors are implemented as `picoObjects`: this means that they are not able to create event channels by themselves and need of the existence of the channel monitor. For a more powerful device, capable of running a standard middleware, the monitor makes no sense, since its functionality is performed by the standard middleware procedures.

Since every actor creates its own specialized event channel to send its events, this approach allows to take under control the message flow, improving at the same time the system scalability.

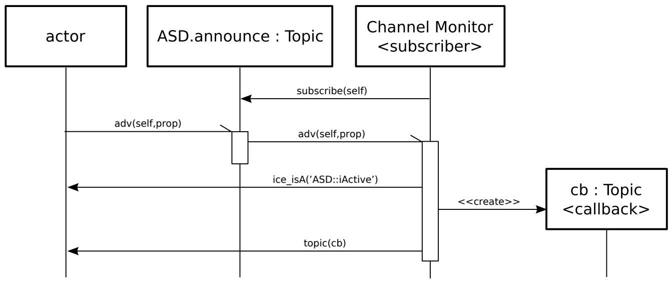

Fig. 1. Sequence diagram for Channel Monitor Service.

4.7 Multi-requests

In WSNs, it is usual that a service requires to query to a certain set of sensors: for example, the service may need to compute the temperature average in a big room with many installed sensors. As a way to simplify this operation, we use reactive actors (see section 4.5).

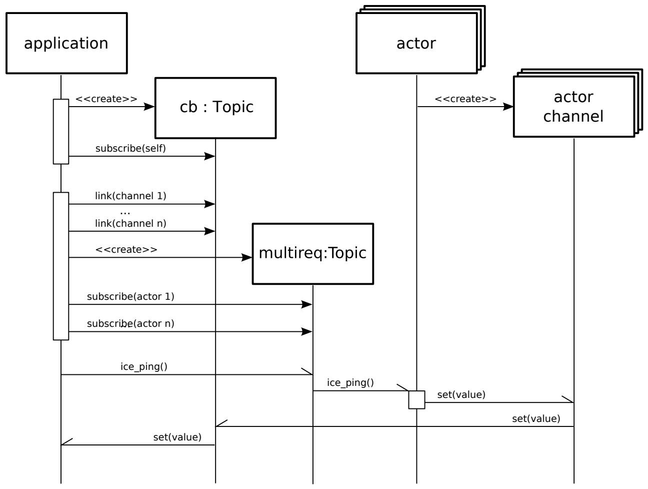

Fig. 2. Sequence diagram for multi-requests.

If a client is interested in the value of a set of sensors, it can create a new event channel. All involved sensors event channels are linked to the new one (if it is known that several nodes share some kind of functional or structural relation the new event channel may be created by default). The clients that are interested in the state of this set of sensors may subscribe to the new channel.

The most efficient way to send the `ice_ping()` to a set of actors is that they hold an additional multicast endpoint. But this is not always possible because it depends on the underlying network technology. For these cases, an alternative solution is proposed (as shown in figure 2).

To make it possible a multiple request, another new event channel is created. All the involved sensors are subscribed to it. This task can be done by an external application, transparently to the nodes. From this moment, when a client sends an `ice_ping()` message, all the nodes receive it.

With the multi-request procedure and thanks to the ICE Storm channels event federation mechanism any external application can configurate its particular *vision of the world* attending to different aspects like functionality, position, security, etc.

4.8 Service Lookup

When an application needs to find a object that provides certain service, the application creates an event channel to be used as "callback" and subscribes to it. Then, the application invokes the `lookup()` method over the *ASD.search* event channel indicating the property values it wants and the callback event channel proxy. The `lookup()` method belongs to the `ASD::iSearch` interface. The application is responsible for the event channel dispose.

```
interface iSearch {
  dictionary<string, ByteSeq> PropDict;
  void lookup(Object* prx, PropDict query);
};
```

The actors (subscribed to the *ASD.search* channel) that match the criterion send an `adv()` message to the channel proxy specified by the application in the `lookup()` message. If other applications or services are interested in the potential replies, they can subscribe to the published channel proxy. A sequence diagram of this procedure is shown in figure 3.

To ensure that actor replies are not sent before others can subscribe to the callback channel, the actor waits for a fixed time before the announcement event is sent. Also, other additional random timeout can be implemented to improve the system scalability.

4.9 Legacy SDP Integration

In large heterogeneous pervasive environments where different networks are deployed (multimedia network, personal body networks, control networks, etc.) it is not likely than only one SDP covers all the different networks. It is also unrealistic to assume that all devices implement just the same SDP. Devices and services from different manufacturers will probably implement several SDPs. Again, a real deployment will require interoperability of several SDPs, at least, for a basic interaction.

We are working on the design and implementation of new procedures that allow a complete interoperability with other SDPs. Looking at the current de

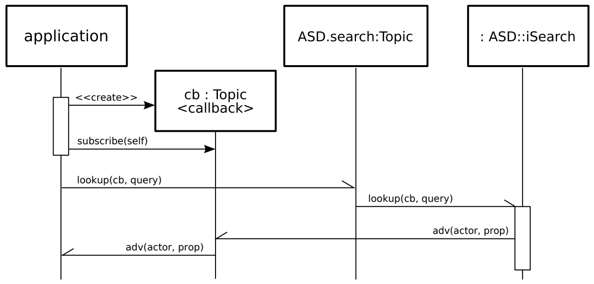

Fig. 3. Sequence diagram for service lookup.

Name of the ASDF Message	Size of Message (in bytes)
Ice::Object::ice_ping	46
IceStorm::TopicManager::create	71
IceStorm::Topic::subscribe	97
IceStorm::Topic::link	91
ASD::iListener::adv	96 (+46 if prop. server)
ASD::iActive::topic	88
ASD::iSearch::lookup	>92 (depends on query)
iByte::W::set	42

Table 1. Size of messages employed in ASDF

facto standard protocols (UPnP, Bluetooth SDP and JINI are being considered) a set of common primitives will be derived so as to make it easy the development of bridges between the ASDF and other SDPs.

Our target is to provide the ASDF with a basic interoperability to, for example, localize and execute services that are offered by a specific WSN node from an UPnP service and without any modification of such a service. To achieve this, we are working on matching the UPnP primitives with the events that can be directly interpreted by the p`picoObjects` that are installed in the WSN nodes.

The choice of the primitives to be implemented and the granularity of the implementation have to be carefully selected and will strongly depend on the SDPs to be integrated.

5 Experimental results

The table 1 shows the size of the messages used in the ASDF protocol, assuming that it has been implemented in ICE. Some of them are standard ICE messages. In the tests, the object identity was 8 bytes long and it used IPv4 endpoints.

In the current prototypes, we are using a 8-bit micro-controller although it is underutilized. Its characteristics are:

– **Model:** Microchip PIC 16LF876A, 10MHz

Type of actor	bytecode	VM	total footprint	RAM used
TCP passive (without `adv()`)	350	333	683	36
TCP passive (periodic `adv()`)	455	411	866	36
TCP reactive (periodic `adv()`)	527	411	938	64
UDP reactive (periodic `adv()`)	368	411	779	64

Table 2. footprint for several `picoObject` nodes (in bytes)

- **Program memory:** 8 KiB
- **RAM:** 368 bytes
- **I/O:** 1 USART, 22 i/o pins, two 8-bit timers and one 16-bit timer.

The table 2 shows the size of several prototype actors. The indicated size includes the complete implementation that runs in the aforementioned microcontroller. No other library or software component is needed. The `picoObject` execution model is composed by a automaton specification (the bytecode) and a small interpreter (a virtual machine, VM) implemented in assembly language. All of them are about two orders of magnitude smaller than any other previous implementation of small embedded standard middlewares.

6 Conclusions

In this paper we have presented a SDP (called ASDF) suitable for low-cost nodes in the WSN field. This SDP allows a *place & play* behavior, so nodes and services can be deployed in a easy and flexible way without any configuration procedure.

Based on a previous work (`picoObjects`), the proposed SDP provides the WSN nodes with an advertisement service by means of *events*. Additionally, it allows external applications to lookup services offered by the WSN nodes.

The design of the ASDF allows incremental addition of functionality according to the device capabilities. Moreover, we have implemented an ASDF prototype using an standard distributed middleware whose common services (event channels, replication, persistence, location transparency, security, etc.) have allowed an easy and reliable implementation.

Due to the interfaces shown in this paper, an application does not distinguish between the advertisement generated by a service resident in a conventional PC or by a node in a WSN. This fact represents a great advantage for quickl development of applications which use WSN services making unnecessary either to integrate in such applications complex WSN specific protocols or to use different programming languages.

In a near future, our work is mainly focused on widening the range of platforms supported by the `picoObject` compiler at same time that we integrate third party services using different SDPs (UPnP and Bluetooth SDP bridges are currently under development) making it possible the real deployment of large heterogeneous pervasive environments under a *place & play* philosophy.

References

1. D. Villa, F.J. Villanueva, F. Moya, F. Rincón, J. Barba, J.C. López. *Embedding a general purpose middleware for seamless interoperability of networked hardware and software components* Grid and Pervasive Computing, GPC 2006, Taiwan May 2006. Lecture Notes in Computer Science 3947.
2. E. Gamma, R.H., R. Johnson, J. Vlissides, *Design Patterns, Elements of Object-Oriented Software*. 1995, Addison-Wesley.
3. F. Stann and J. Heidemann. *BARD:Bayesian-assisted resource discovery in sensor networks* in Proceedings of the IEEE Infocom, 2005.
4. Timmons, N.F.; Scanlon, W.G., *Analysis of the performance of IEEE 802.15.4 for medical sensor body area networking*, IEEE SECON 2004, October 2004
5. J. Lundquist, D. Cayan, and M. Dettinger., *Meteorology and Hydrology in Yosemite National Park: A Sensor Network Application*, Information Processing in Sensor Networks (IPSN), April 2003
6. A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, *Wireless Sensor Networks for Habitat Monitoring*, WSNA'02, September 2002
7. S. Tilak, K. Chiu, N.B. Abu-Ghazaleh and T. Fountain, *Dynamic Resource Discovery for Wireless Sensor Networks* IFIP International Symposium on Network-Centric Ubiquitous Systems (NCUS 2005)
8. Microsoft, *UPnP Device Architecture v1.0* Available at http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.
9. E. Guttman and C. Perkins and J. Veizades and M. Day, *Service Location Protocol, Version 2*, RFC 2608, 1999.
10. Bluetooth SIG, *Specification of the Bluetooth System v2.0*, available at http://www.bluetooth.org. November, 2004.
11. U.C. Kozat and L. Tassiulas. *Service Discovery in mobile ad-hoc networks: an overall perspectiva on architectural choices and network layer support issues* Journal on Ad-hoc Networks, 2004.
12. F. Sailhan and V. Issarny. *Scalable Service Discovery for MANET* Proceedings of the 3rd IEEE conference on Pervasive Computing and communications, 2005.
13. C. Campo and M. Munoz and J.C. Perea and A. Marin and C. Garcia Rubio, *PDP and GSDL, a new service discovery middleware to support spontaneous interactions in pervasive systems*, Pervasive Computing and Communications Workshop, 2005.
14. M. Kuorilehto, M. Hannikainen and T. Hamalainen, *A Survey of Application Distribution in Wireless Sensor Networks* EURASIP journal on Wireless Communications and Networking 2005:5,pp 774-788.
15. P. Baronti, P. Pillai, V. Chook, S. Chessa, A. Gotta, Y. Fun Hu, *Wireless Sensor Networks: a Survey on the State of the Art and the 802.15.4 and ZigBee Standards* Technical Report ISTI-2006-TR-18, Istituto di Scienza e Tecnologie dell'Informazione del CNR, Pisa, Italy, November 2006, pp.41.
16. Sun Microsystems, *Jini Architecture Specification*, ed. 1.2, available online at http://www.sun.com/,
17. ZeroC, Inc., *ICE Home Page*, available online at http://www.zeroc.com/,
18. ARCO Group, *PicoObject Web demostration example*, available at http://arco.inf-cr.uclm.es/marisa.html.en