ASDF: An Object Oriented Service Discovery Framework for Wireless Sensor Networks

D. Villa
David.VIlla@uclm.es
F. J. Villanueva
Felix.Villanueva@uclm.es
F. Moya
Francisco.Moya@uclm.es
F. Rincón
Fernando.Rincon@uclm.es
J. Barba
Jesus.Barba@uclm.es
J. C. López
JuanCarlos.Lopez@uclm.es
Dept. of Technology and Information Systems
University of Castilla-La Mancha
School of Computer Science. 13071 - Ciudad Real. Spain

***Abstract*—**

Purpose: **This paper presents a new Service Discovery Protocol (SDP) suitable for Wireless Sensor Networks (WSN). The constraints imposed by ultra low-cost sensor and actuators devices (basic components of a WSN) are taken into account to minimize the overall footprint.**

Design/methodology/approach: **It is based on the lightweight WSN communication model used by `picoObjects`, a tiny implementation of the distributed object concept. We consistently follow the same design criteria aiming at minimal overhead for devices and communication protocols. In spite of its simplicity it is powerful enough to deploy a valuable set of services.**

Findings: **This approach provides a remote interface that client applications can use without knowing where the service is implemented (platform and location independence).**

Research limitations/implications: **The future work is mainly focused on integrating third party services using different SDPs, making it possible the real deployment of large heterogeneous pervasive environments.**

Practical implications: **Designers may change the underlying SDP model (e.g. centralized vs. distributed) without affecting applications by just tweaking configuration settings.**

Originality/value: **Embedded devices can participate in the Service Discovering Procedure providing their own services by means of standard distributed objects. Besides, the protocol is suitable for any kind of dynamic networked system.**

Keywords: **Service Discovery, Resource Discovery, Wireless Sensor Networks, Object Oriented Middleware**

Paper type: **Research paper**

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)

I. Introduction

Wireless Sensor Networks (WSNs) are becoming the key component in any pervasive environment. They usually implement the interaction with the physical world, both the ability to measure physical magnitudes and the ability to react to changes in those magnitudes by driving an actuator. 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 are meant to measure the value of a given physical magnitude (e.g temperature, humidity, smoke, etc.) while actuators are able 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. Finally the network interface offers wireless network connectivity.

Flexibility and quick deployment (mainly due to their wireless interface) are the characteristics that make WSNs a nice solution for multiple applications such medical [5] or meteorology [6] applications, habitat monitoring [7], etc. In the short term we envision a pervasive environment plenty of heterogeneous WSN nodes offering a wide variety of services, ranging from the most basic (supported by either individual nodes or the whole network) to the most complex (broadly distributed ambient intelligence services).

However, the flexibility in the physical deployment of WSN (wiring is unnecessary) did not match its software counterpart. We believe that the deployment of a WSN should also care of minimizing the configuration. With the service discovery protocol (SDP) described in this paper, a WSN node gains the

ability to announce its services and offer the possibility to use them without any previous configuration procedure.
The proposed SDP:

- Allows very low-cost nodes to be deployed in an easy and incremental way (following a *Place & Play* philosophy).
- Allows applications to discover and use the services offered along a WSN (specially convenient in mobile applications).
- Is designed for heterogeneous WSNs where different nodes implement different functionality using different technologies.

The SDP described in this paper is based on our previous work on `picoObjects` [1]. This approach achieves a high degree of interoperability with standard distributed object oriented middlewares, and also provides the capability 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 [19] (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 remaining of this paper is organized as follows. Section II explains some previous works on SDPs. In section III the `picoObject` approach is briefly summarized. Section IV is devoted to our SDP in detail. In section XV the prototype we used to validate our proposal is briefly described. Finally we draw some conclusions and outline some future work.

II. Related work

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

Some currently broadly used SDPs such as UPnP [9], Jini lookup service [18], Bluetooth SDP [11] or SLP [10] are considered *de facto* standards. The evolution of fields such as ambient intelligence, 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 [15] do not assume such constraints in their design.

Recent works have proposed SDPs for new technologies like mobile ad-hoc networks [13][14]. 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 [16][17] are working on prototypes which need to be reduced in terms of cost (therefore probably in resources) for an eventual massive introduction in the market.

In [8] 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. The CH will also answer SQL queries in place of its cluster sensors. The CH is selected among 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 IV, provides a way to incrementally add functionality to the nodes, so that 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 widespread prototyping platforms like MICA, MicaZ, RockWell WINS, etc.

Finally, in [4] an homogeneous sensor network (all the nodes have the same functionality) resource discovery protocol is proposed, centered in the optimization of the flooding process by taking advantage of historical queries [8]. Our work assumes that a WSN is composed 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 did not face 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.

III. `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 `picoObject` 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 [20].

IV. Introduction to ASDF

We have defined an ultra lightweight object-oriented service discovery protocol, called ASDF (*Abstract Service Discovery Framework*), which provides several valuable features such as:

- An easy way for device announcement.
- Extensibility and scalability.
- Interaction with legacy SDP.
- Seamless integration with standard middlewares.
- Auto-configuration for devices (in order to get a *place & play* behavior).

The ASDF is designed keeping in mind minimal devices. For example, the protocol allows the nodes to announce themselves to the network using simple messages completely compatible with the distributed object middleware. In spite of this, the protocol is very scalable and may be applied to more powerful devices.

V. Event Channels

Our protocol uses extensively the standard *event* service provided by the distributed object middleware. This makes it possible to easily decouple all involved elements. The event channel is a direct implementation of the *observer* [3] design pattern (also known as *publish-subscribe*).

The IceStorm service (ZeroC ICE *event channel*) is able to employ several transport protocols at same time (TCP, SSL, UDP and multicast UDP are all implemented in the stock version). This is entirely transparent for objects. Even on a single shared channel each publisher or subscriber may choose the protocols to use individually.

However, connecting too many nodes to the same event channel may raise scalability concerns. 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. There is also a simple mechanism to specify limits or priorities to event propagation.

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

VI. 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 (a node/device exposing its functionality by means of an object interface) is connected or returns from a sleeping state, the node sends an announcement message (`adv()`) to a specific event channel called ASDA: *ASD Announce*. Optionally, these announcements may 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 holds the node properties (see section VIII). The next listing shows 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 Params Size: 16
        Input Params Encoding: 1,0 - Encap. params (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 ASDA (see figure 1). 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`.

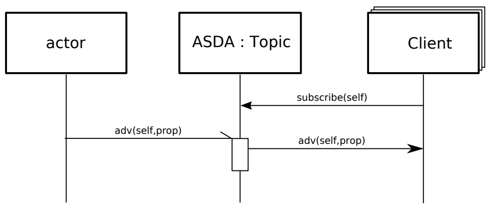


Fig. 1. Sequence diagram for ASDA channel interaction

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".

VII. Service Description

The model for service description is defined in an ontology. The MIS *Model Information Service* provides access to his ontology by means of its `Metamodel` interface. Any peer may obtain the set of valid attributes of a specific class or the relations between classes.

Besides MIS is also responsible for legacy SDP integration (see section XIV).

```
module MIS{
  interface Metamodel{
    list getClasses();
    list getPropertyNames(string classID);
    string getValueType(string classID, string propertyID);
    list getLegalValues(string classID, string propertyID);
  };
};
```

The description of a specific service requires two items:

The property set
The properties represent constraints of a specific actor. That is, a temperature sensor holds several properties: maximum and minimum values, precision, localization, etc. The next section describes properties in detail.

The interface
This is defined in Slice language and corresponds to a single class of the ontology[1]. These are non-behavioral interfaces. Their purpose is to identify a type of service and any object may inherit from one or several of these interfaces. The following listing shows some of these type of service interfaces. Section XIII shows how to use this information in the service discovery process.

```
module Service {
  interface PresenceSensor{};
  interface TemperatureSenser{};
  interface MotionSensor{};
  [...]
};
```

VIII. Properties

As mentioned above, the parameter `prop` in the `adv()` message is an object proxy for a "property context". This property context allows the clients to access the actor properties. There are several alternatives here:

- The argument `prop` can be a null proxy when it is not necessary or the actor does not have any property.
- If the device has enough computing resources, its property context may be implemented in the device itself. In this case, both `adv()` arguments, `prx` and `prop`, may point to the same object. This service description storage option is known as *Unstructured Distributed* [12].
- The proxy `prop` may also point to a remote object in a different location. This allows to implement collective property contexts (called *Property Servers*) for many actors whose properties are stored outside the actor, perhaps in a full blown database. Even a single servant may be able to dispatch method invocations for many objects using a "default servant" strategy. This service description storage option supports centralized and hybrid models.

The *Property Context* implements the `iProperties` interface:

```
module ASD  {
  dictionary<string, Object> PropDict;

  module iProperties {
    interface R {
      idempotent Ice::StringSeq keys();
      idempotent PropDict getp(Ice::StringSeq keys);
    };
    interface RW extends R {
      void setp(PropDict dict);
    };
  };

  class boolValue { bool value; };
  class byteValue { byte value; };
  ...
}
```

[1]A simple script takes an ontology description (.OWL file) and generates Slice declarations.

Properties are specified by means of key, value pairs. The key is just a text string naming the property. The value is an object of a primitive type such as `boolValue` or `byteValue`. We are currently adopting the Standard Property Service Specification by OMG [2] because it provides a more feature-full interface.

We should emphasize that actor properties are considered optional (not required) information. It is useful for administration, configuration and monitoring tools but it doesn't affect the system basic functionality. System services never depend on property values or their availability.

IX. Basic interface for actors

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

```
module iBool {
  interface R { idempotent bool get(); };
  interface W { void set(bool v, Ice::Identity oid); };
};

module iByte {
  interface R { idempotent byte get(); };
  interface W { void set(byte v, Ice::Identity oid); };
};

module iFloat {
...
```

X. Interaction model for actors

Depending on how the application interacts with actors, there are four basic types of actor behavior:

Passive
To get the state value of a passive sensor, the client needs to invoke explicitly the `get()` method of the actor and then it 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
This is a special case of an active sensor but it sends the `set()` event when a change occurs in its state.

Reactive
A reactive sensor is also an active sensor which sends `set()` events only if a client invokes its standard `ice_ping()` method. The `ice_ping()` standard functionality has been extended so that

when this method is invoked, the actor will send an event to the pre-defined event channel to publish its state, in addition to the conventional `ice_ping()` behaviour.

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` 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) that the actor must use to publish its events.

XI. 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 2):

- Using the middleware introspection features, it verifies that the new actor is actually an active actor (implements the `iActive` interface).
- 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.
- After creating the corresponding event channel, the monitor invokes the `topic()` method of the actor with the proxy for the new event channel as the argument.

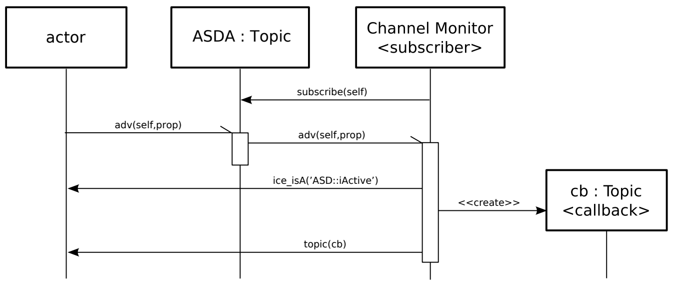


Fig. 2. Sequence diagram for Channel Monitor Service.

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 therefore they need an external 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 a fine-grain control of the message flow, improving at the same time the system scalability.

XII. Multi-requests

In WSNs it is quite a common situation when a service needs to query a certain set of sensors. For example, a service may need to compute the average temperature in a big room with many sensors installed. As a way to simplify this operation, we use reactive actors (see section X).

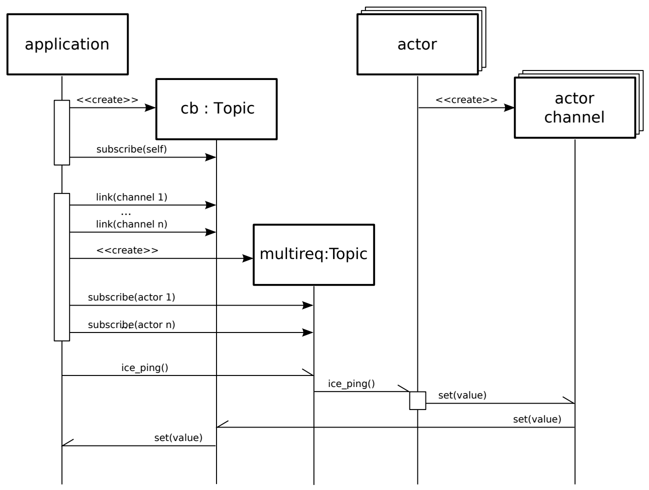


Fig. 3. Sequence diagram for multi-requests.

If a client is interested in the value of a set of sensors, it should create a new event channel. The event channels associated to all the involved sensors are linked to the new one. If it is known that several nodes share some kind of functional or structural relation this new event channel may already be created by default. The clients that are interested in the state of this set of sensors may subscribe to this 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 3).

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

With the multi-request procedure and thanks to the ICE Storm event channel federation mechanism any external application can configure its particular *view of the world* according to different aspects such as functionality, position, security, etc.

XIII. Service Lookup

When an application needs to find an object that provides a given service, the application creates an event channel to be used as "callback" and subscribes to it. Then, the application invokes the `lookup()` method on the ASDS (*ASD Search*) event channel with the following information:

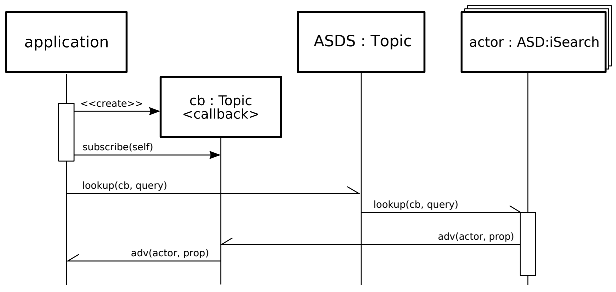


Fig. 4. Distributed service lookup.

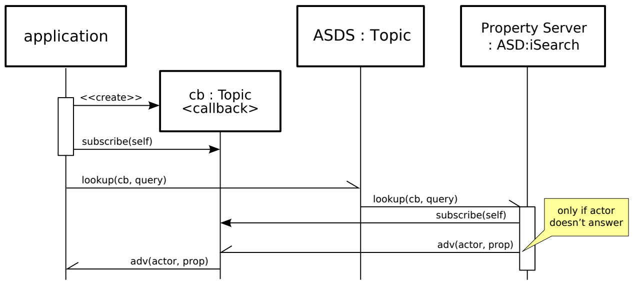


Fig. 5. Property Server as a Service Directory.

- The service interface (the ontology class).
- A set of properties that objects must match.
- The proxy of the callback event channel. The application is responsible for releasing the callback event channel when it is not needed.

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

The actors subscribed to the *ASDS* channel that match the seach criteria will send an `adv()` message to the channel proxy specified by the application in the `lookup()` message. If other applications or services are also interested in the potential replies, they can also subscribe to the published channel proxy. A sequence diagram of this procedure is shown in figure 4.

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

A. *Service Directory*

Given the type and the amount of information contained in the *Property Servers* they can play an active role in service discovery. The Property Server may subscribe to the ASDS channel to receive all of the `lookup()` events. It then evaluates the search criteria for all its known actors generating announcements for every matching actor (see figure 5).

This behaviour may even coexist with the distributed directory service described before. A Property Server just needs to wait for a given timeout before it produces an answer. If a matching service is able to respond itself then the Property Server will skip it. In a worst case scenario there will be two identical announcements for each actor but since announcements are idempotent there will be no negative consequences from the functional point of view.

B. *Hierarchical Service Lookup*

Service discovery may not scale up indefinitely. As in most of the ASD protocol it was designed for environments with a predictable amount of devices. But sometimes it may be needed to look for a service in the upper level of the hierarchy. Then we rely on IceGrid, another standard service included in the ZeroC ICE middleware.

Among the many features included in IceGrid (replication, load balancing, implicit activation, ...) we are mainly concerned with the following two:

IceGrid/Locator
: It allows contacting remote objects indirectly through an *indirect proxy*. These objects must be previously registered into another component called IceGrid/Registry.

IceGrid/Query
: This service offers an interface for querying the Registry database of well-known objects. The following listing shows some methods included in this interface:

```
interface Query {
    Object* findObjectById(Ice::Identity id);
    Object* findObjectByType(string type);
    Ice::ObjectProxySeq findAllObjectsByType(string type);
    Ice::ObjectProxySeq findAllReplicas(Object* proxy);
};
```

Therefore IceGrid/Query is used as a service directory since it allows queries by service type. Nonetheless this is only one side of the problem. We also need to provide a specialized *Locator* object with the ability to redirect requests to the upper hierarchical level when needed. Besides we must match against the set of properties requested in the *lookup()* invocation.

From the point of view of clients and actors this new component called *ExternalLocator* is fully transparent and its behaviour is similar to what is described in section XIII-A. The only noticeable difference is that while ExternalLocator works as a two-level hierarchical directory 6, the *PropertyServer* is a centralized directory.

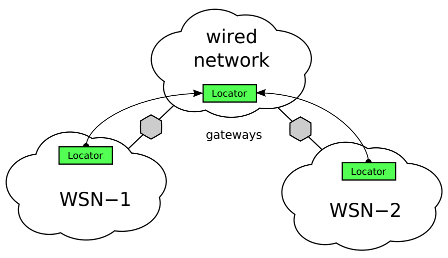


Fig. 6. Service lookup delegation to upper domain level

C. Support for 'pull' discover

The advertisement model described in VI is known as *Passive Directory* or *Pure Push Model*, that is, the actors offer their services explicitly and the clients do not have any other way to detect them. This is not the most convenient model for WSN, where power consumption is a very important issue. It would be better if the actors would only send advertisements when are connected or optionally with a low periodicity.

Therefore the ASD search service provides a generic environment discovery mechanism, supporting a hybrid Push/Pull model. In these cases, the clients may send a `discover()` message to the ASDS topic to query updated advertisements. From the server side there are two different alternatives:

Disributed
: The actors are ASDS subscribers and send `adv()` messages when they receive the `discover()` request (see figure 7).

Centralized
: A specialized service called *Cache Service* is an ASDA subscriber and it store all the advertisement information. This service is also an ASDS subscriber. When it receives `discover()`, it sends `adv()` messages for each previously advertised actor (see figure 8).

In both alternatives, the clients receive conventional actor advertisements by means of ASDA channel.

The pure Push Model may be achieved when the *Cache Service* knows the objects in advance. However, that situation is not convenient for dynamic environments such as WSNs.

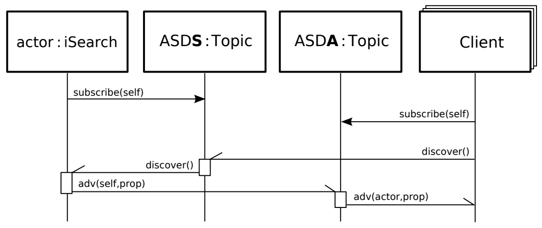


Fig. 7. Distributed discover

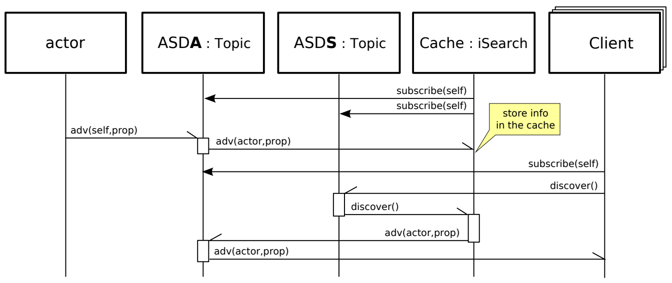


Fig. 8. Centralized discover: cache

XIV. 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 that a single 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 among other SDPs. Looking at the current *de facto* standard protocols (UPnP, IETF SLP, Bluetooth SDP and Jini are being considered) a set of common primitives will be derived in order to ease the development of bridges between the ASDF and other SDPs (see table I).

With these kind of semi-automatic bridges ASDF may be used to locate and use 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 `picoObjects` that are installed in the WSN nodes.

The choice of the primitives to be implemented and the granularity of the implementation needs to be carefully analyzed and will strongly depend on the SDPs to be integrated. Each external SDP requires a semi-custom `SDP Gateway` which translates SDP primitives to and from ASDF primitives.

Another topic related to SDP legacy integration is how to match the ASDF nomenclature with a third party platform nomenclature. This correspondence is a responsibility of the the MIS service. For example, in UPnP service templates the string "urn:schemas-upnp-org:service:TemperatureSensor:1" identifies a temperature sensor service in the UPnP domain. Our model information service defines the class *TemperatureSensor* where a especific instance has been defined for the UPnP temperature service.

With this model, when an UPnP `SDP Gateway` identifies a native lookup request for any service, this gateway asks the MIS service for the correspondence between the UPnP service and our ASDF naming scheme. For example the UPnP service "urn:schemas-upnp-org:service:TemperatureSensor:1" is translated by the MIS into a TemperatureSensor string and the `SDP Gateway` will invoke an ASDF lookup to find the TemperatureSensor service.

Finally, it is also necessary to instantiate a specific service gateway between the UPnP service and `picoObject` clients. This instantiation is also implemented by the `SDP Gateway` when it receives an answer to the request for a specific service.

XV. Experimental results

Table II 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 8-bit microcontrollers although it is underutilized. These are the main technical characteristics:

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

ASDF	UPnP SSDP	JINI	BlueTooth	IETF SLP
Search Directory	No directory	Search Lookup Service	SearchRequest SearchResponse	SrvTypeRqst
Find service at diretory	No directory	Lookup Service	Search Req/Resp ServiceAttributeReq/Resp	SrvTypeRqst/AttrRqst
Find service (no directory)	Multicast SSDP	Not supported	ServiceSearchReq/Resp ServiceAttributeReq/Resp	SrvTypeRqst/AttrRqst
Advertisements	Mulicast	SSDP Announce protocol (Lookup Service)	Register service in the SDP server	DAAdvert/SAAdvert
Registration	Advertisements	Registration in a Lookup Service	Register service in the SDP server	SrvReg
Subscription	Subscription to GENA events	Not supported	Not supported	Not supoorted
Renew leases	Application policies not supported	Lease Renewal Manager	Not supported	Not supported
Disconnect	bye bye SSDP message	Remove/Cancel Leasing in Lookup Service	Not supported	Not supported

TABLE I
UPnP, Jini, BlueTooth SDP, IETF SLP and ASDF comparison

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 II
Size of messages employed in ASDF

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 III
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.

Table III shows the size of several prototype actors. The size shown includes the complete implementation that runs in the aforementioned micro-controller. 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.

XVI. 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 in several ways.

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 quick development of applications which use WSN services making unnecessary to integrate complex custom WSN protocols.

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 with 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] Object Management Group, *Property Service Specification* , April 2000. Available in `http://www.omg.org/`, document id: 00-06-22.

[3] E. Gamma, R.H., R. Johnson, J. Vlissides, *Design Patterns, Elements of Object-Oriented Software*. 1995, Addison-Wesley.

[4] F. Stann and J. Heidemann. *BARD:Bayesian-assisted resource discovery in sensor networks* in Proceedings of the IEEE Infocom, 2005.

[5] 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

[6] 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

[7] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, *Wireless Sensor Networks for Habitat Monitoring*, WSNA'02, September 2002

[8] 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)

[9] Microsoft, *UPnP Device Architecture v1.0* Available at http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.

[10] E. Guttman and C. Perkins and J. Veizades and M. Day, *Service Location Protocol, Version 2*, RFC 2608, 1999.

[11] Bluetooth SIG, *Specification of the Bluetooth System v2.0*, available at http://www.bluetooth.org. November, 2004.

[12] Raluca Marin-Perianu, Pieter Hartel, Hans Scholten, *A Classification of Service Discovery Protocols*, June 2005.

[13] 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.

[14] F. Sailhan and V. Issarny. *Scalable Service Discovery for MANET* Proceedings of the 3rd IEEE conference on Pervasive Computing and communications, 2005.

[15] 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.

[16] 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.

[17] 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.

[18] Sun Microsystems, *Jini Architecture Specification*, ed. 1.2, available online at `http://www.sun.com/`,

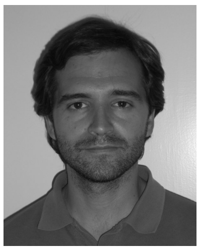
[19] ZeroC, Inc., *ICE Home Page*, available online at `http://www.zeroc.com/`,

[20] ARCO Group, *PicoObject Web demostration example*, available at `http://mauchly.inf-cr.uclm.es/wiki/index.php/Arco_Projects`

David Villa received his MS degree in Computer Engineering from the University of Castilla-La Mancha in 2002. Since then he works as a Teaching Assistant at the University of Castilla-La Mancha (UCLM). He is currently pursuing the PhD degree in Computer Science from UCLM. His current research interests include heterogeneous distributed systems, and distributed embedded system design.

Félix J. Villanueva received the Computer Eng. Diploma from the University of Castilla-La Mancha (UCLM) in 2001. In 1998 he joined the Computer Architecture and Networks Group at UCLM where he is now working as Teaching Assistant. He is currently pursuing the PhD degree in Computer Science from UCLM. His research interests include wireless sensor networks, ambient intelligence and embedded systems.

Francisco Moya received his MS and PhD degrees in Telecommunication Engineering from the Technical University of Madrid (UPM), Spain, in 1996 and 2003 respectively. From 1999 he works as an Assistant Professor at the University of Castilla-La Mancha (UCLM). His current research interests include heterogeneous distributed systems and networks, electronic design automation, and its applications to large-scale domotics and system-on-chip design.



Fernando Rincón completed his graduate studies in Computer Science at the Autonomous University of Barcelona in 1993. In 2003 he obtained the PhD degree from the University of Castilla-La Mancha, where he is currently an Assistant Professor. His research interests include System-On-Chip integration, Hw run-time reconfiguration and Heterogeneous Distributed Systems.

Jesús Barba received the Computer Engineering Diploma from the University of Castilla-La Mancha (UCLM), Spain, in 2001. In 1998 he joined the Computer Architecture and Networks Group at UCLM where he is working as Teaching Assistant with the Department of Information and Systems Technology from 2001. He is currently pursuing the PhD degree in Computer Science from UCLM. His research interests include SoCs, HW/SW integration and embedded distributed systems.

Juan Carlos López received the MS and PhD degrees in Telecommunication (Electrical) Engineering from the Technical University of Madrid (UPM) in 1985 and 1989, respectively. From Sep 1990 to Aug 1992, he was a Visiting Scientist in the Department of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA (USA). His research activities center on computer-aided design of integrated circuits and systems. His work is focused on algorithms for automatic synthesis, co-design and embedded computing. From 1989 to 1999, he has been an Associate Professor of the Department of Electrical Engineering at UPM. Currently, Dr. López is a Professor of Computer Architecture and Dean of the School of Computer Science at the University of Castilla-La Mancha.