A development model supporting integrative object oriented middlewares for sensor and actuator networks

C. Martín, D. Villa, F. J. Villanueva, F. Moya, M. J. Santofimia and J. C. López
Computer Architecture and Networks Group
University of Castilla La-Mancha
Ciudad Real, Spain

Abstract— *Today there are many proposals of middlewares for sensor and actuator networks (SAN). Each of them introduces programming paradigms for creating distributed applications. There are middlewares with different philosophies in terms of architecture and organization (database, clustering, etc.). From the standpoint of the programmer these considerations influence the development process and, in most cases, determine the programming paradigm.*

Although these middlewares are so different, there is a common issue: they assume the SAN is logically separated from the main network. Thus, it forces the programmers to think a lot on specific stuff for the SAN and provides a dissociated vision of the whole system.

In this paper we propose a new development model for SAN based on standard object oriented middlewares. Moreover, we provide a language as a new programming paradigm so that it is possible to have a homogeneous view of the whole system (including main network and the SAN).

Keywords: sensor and actuators networks, wireless sensor networks middleware, integration, development model

1. Introduction

Currently, sensor and actuator networks (SAN) have been adopted for many information systems: monitoring systems, industrial control processes, medicine and many others. There is a huge set of different hardware implementations of the SAN nodes, and each manufacturer provides software for creating applications on their own platforms.

The scientific community is aware of the difficulty in integrating different hardware and software nodes in a single abstract networked system. Heterogeneous distributed systems may be a solution for this issue and some approaches use middlewares as integration platform. There are many middleware based solutions [1] for building applications in SAN but most of them are non-compatible with the main network (composed by servers and other more powerful resources than SAN nodes).

For this reason, the software development process and tools used for building applications in a SAN is separated from the remaining of the distributed system. From the development process viewpoint, object oriented middlewares are desirable because they provide high abstraction level to the designers and programmers. Moreover, some commercial middlewares like CORBA [2], ZeroC-Ice [3], or Java RMI [4] offer very valuable built-in services that may be used in SAN (such as location transparency, event channels, etc.).

Creating applications by using the conventional vendor software in a heterogeneous context may be a hard task. Moreover, this kind of devices are microcontroller based and have a few Kwords of program memory. So, the reduced hardware resources prevent loading conventional middleware software for being integrate in the heterogeneous system. In [5], [6], it highlights the need to integrate sensor networks in the whole system.

In this work we propose a new development process for low-cost and resource limited devices so that it is possible to integrate SAN in heterogeneous distributed systems. We bring a convenient development methodology and we purpose a distributed object oriented programming paradigm for devices in SAN that permits hardware abstraction and high level programming. In this paradigm, it is assumed the SAN *is part* of the whole system and there is no logical difference between main conventional system network and sensor networks.

In section 2, we discuss the related work and the concepts on we have based. In section 3 it is exposed the development process model and in section 4 we show an application example in which are explained the components of the designed concepts and tools. Finally, in the sections 5 and 6, it is offered results of different applications developed and the conclusion, respectively.

2. Related work

Each SAN middleware provides a development model for programmers based in their own architecture and technology. Following, it is described the current programming paradigms for sensors and actuators networks (SAN) based in middleware. Most of them are focused on wireless sensor networks (WSN) due to the specific needs of this kind of networks [1]:

- *Database*: based in a "virtual" database, TinyDB [7] is a good representative. It provides an API for Java

and nesC programs (for TinyOS [8]). The data access procedure is expressed in SQL-like queries which the middleware process.

- *Macro-programming*: the programmer describes the whole SAN behavior by using a high-level programming language. Then, the runtime deploys the code to individual nodes. Maté [9], [10], Magnet [11] and SensorWare [12], [13] are example of this paradigm.
- *Clusters*: distributed system consists of a set of node groups (cluster) where each group has a *head* node. This node is able to route, filter and collect data. For example, SINA [14] or DSWare [15] are sensor middlewares based on clustering and offer a database programming paradigm.
- *Virtual machine*: this approach install a virtual machine on each sensor node for executing a certain bytecode. Despite the cost involved, it may be interesting using a virtual machine for reach code portability. From programmer viewpoint, virtual machine hides the concrete platform and communication issues.
  Maté defines an abstract instruction set architecture (ISA) for developing applications in sensor nodes. However, this ISA is not abstract enough since it provides closed platform instructions.
  SensorWare and WSP [16] are examples of the virtual machine approach.

All this solutions assumes that SAN entities are separated logically from the rest of the distributed system. However, there is an integrated solution proposed by Moya et al. [17] that use standard and generic middlewares.

This approach describes the concept of *picoObject* [18], [19], that is, a state machine described in FSM language for parsing specific protocol messages, running on resource-constrained platforms and serving one or more distributed objects. The picoObjects may be integrated in object oriented middlewares implementing the inter-ORB protocol (i.e. GIOP). The state machine code is executed by a FSM virtual machine implemented in a platform-dependent language and technology.

picoObjects have the advantages of the virtual machine paradigms and may be used to integrate SAN in the distributed system. The sensor nodes may be seen on the same way as any other object on the network. However, writing a FSM that implements even basic services is a heavy and error prone task because its low-level orientation.

3. Development Model

We have defined a development model for SAN nodes based on the use of object oriented middlewares and event-driven automata like picoObjects. Figure 1 shows the four main actors involved in this model and their tasks and responsibilities are bounded. In the figure, server application resides in the sensor node (labeled as picoObject) and client application may be implemented for any other node of the

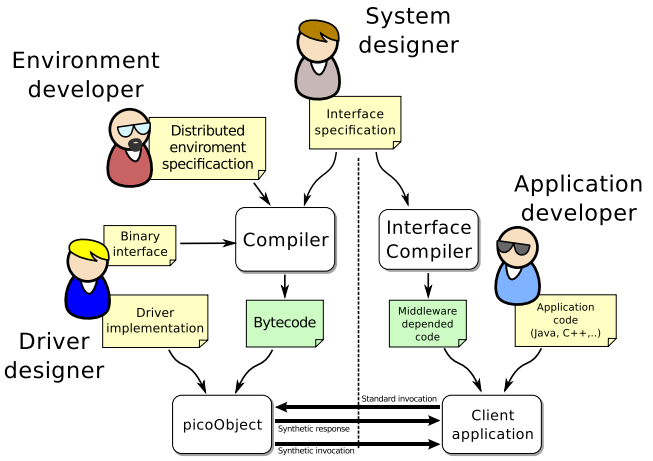

Fig. 1: Proposed development model

distributed system (server, router, PC or either other which is not included in the SAN).

On one hand, the server side is composed by a sensor node. We assume that in this node runs a picoObject virtual machine serving one or more distributed objects. On the other hand, we consider a client side where a tipical distributed application may be used.

Before the client and server processes start it is needed to definite the distributed object interfaces. Thus, the *system designer* is responsible of defining the interfaces implemented by the distributed objects and designing the system behavior. In systems based on CORBA, the designer should define these object interfaces using IDL code. Other object oriented middlewares offer similar languages like Slice (for ZeroC Ice middleware) and Java (for Java RMI).

This information is shared between the server and client applications, thus client knows the invokable server methods and server knows the methods that it must implement. So, once the interface information is defined server and client design processes may start concurrently.

First, the server development process is described:

- *Environment developer*: the environment developer have to define the distributed environment in which the sensor node will be included. In general, the programmer describes all useful information for the target node:
  - *Objects*: describe the objects whose are part of the system. Any distributed object must to implement one interface (at least). The environment developer may use the interface definitions provided by the system designer.
  - *Nodes*: a node may be seen as a set of distributed objects accessible by an endpoint. It is necessary to describe the nodes of the distributed system. This definition must include the target node and those nodes that have functional relationship with it.

Note that hardware implementation of the node does not matter. These nodes are abstract and may represent different hardware technology nodes.
  - *Relations*: implement the relationship between nodes and objects. In other words, it is necessary to define which objects are registered in which node.

  Besides this, the environment developer should describe the object behavior of the target node. As it will be shown, this description may be implemented by using invocations between objects and other high abstraction level structures.
- *Driver designer*: so far, it has been defined the interfaces of the distributed objects, the scenario in which the target node will be immersed and its behavior within the distributed system. However, it is needed to implement the methods effectively. This is also necessary when building a server application in object oriented middleware, that is, the driver designer has to build the platform-dependent code which implements the methods of the sensor node resident objects. This code concerns OS calls, hardware access and other low-level tasks. This software may be develop by the driver designer or be provided by the vendor.

  In any case, the driver designer should provide a *binary interface* which be useful to join the behavior description and the final implementation.
- *Compiling*: with all described information bellow, a compiler can generate a interpretable bytecode for a virtual machine. This bytecode represents all the defined objects (their interfaces and other attributes) and the event-driven behavior of the automata.

  The virtual machine (picoObject), the associated bytecode and the driver implementation are the components whose should be loaded in the physical device. Thus, sensor node is ready for being integrated in the distributed system on the same way as any other distributed object.

The *application developer* has the same tasks as if were to develop a tipical client for middlewares like CORBA or ZeroC-Ice. The designer should use a compiler which translate the interfaces specification in a certain language (IDL for CORBA and Slice [3] for Ice) into an implementation in the same language of the client application code. Normally, these translators are provided in middleware distributions.

Once the interface code is translated, application developer just add the specific application code and no more.

4. IcePick, SIS and ipkc compiler

We have developed several tools to be used in the development model described below. The main tools are:

- *IcePick*: is a language designed for describing the distributed system, the relations between nodes and objects and the target node behavior. IcePick is an hybrid language. On one hand, it is a declarative language for describing the distributed scenario. On the other hand, IcePick allows programmer to define the sensor node behavior in an imperative way by using local and remote invocations.

  The behavior is described based on an event-driven automata, so the invocations are grouped in different kinds of *triggers*.
- *SIS*: the *Servant Interface Specification* is a language for building the binary interface (provided by the driver designer). In SIS files, it is possible to define the binary interface with the follow entities:
  - *Events*: automata has an event-driven behavior. SIS provides structures for declaring the basic interface with the event implementation.
  - *Remote methods*: every method declared at any object interface is accessible remotely. SIS provides structures for establishing correspondence between the methods used in IcePick and their implementations.
  - *Local methods*: it is possible that driver designer offers methods that are not declared as part of the object interface, but may be used by the environment developer as locally callable method. SIS provides structures that allow to specify the binary interface with this kind of methods.
- *ipkc*: is a compiler that accepts the interface specification, an IcePick and a SIS file as input and generates bytecode for a selected virtual machine. Figure 2 shows the ipkc compiler structure. The shown compilation process is for a TinyOS based device, but it is possible generate code for other virtual machine implementa-

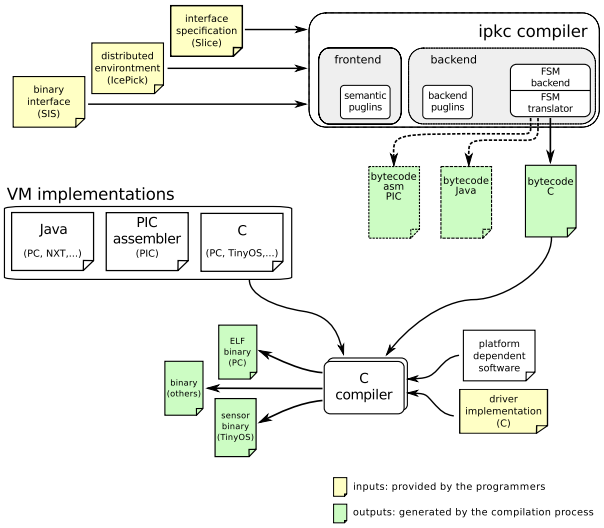

Fig. 2: ipkc compiler process

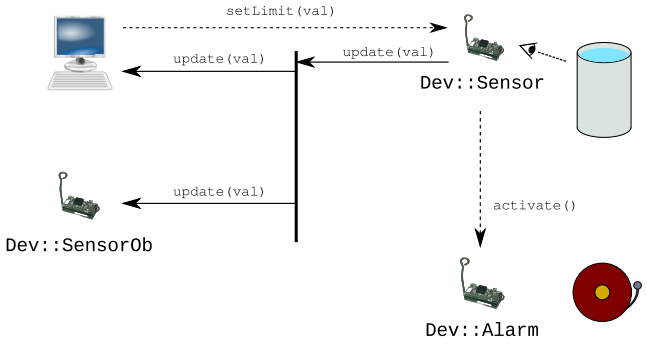

Fig. 3: The tank level controller

tions.

In general, the modular architecture of the compiler allows extensibility in the compilation process. The plugin system allows to extend the semantic level of IcePick, so the programmer may adapt IcePick to a concrete context problem. An IcePick semantic extension may require a specific code generation process. For this reason, ipkc backends may delegate the generation of the concrete context code to a backend plugin.

In the figure, Slice is used as interface specification language and generates bytecode for a C implemented VM. Thus, it is possible generate picoObjects for the Ice middleware. Although ipkc compiler may integrate new languages as IDL [2], and also other code generators as the CORBA picoObjects backend [17].

For better understanding, an application example developed using the described tools is shown below. As a simple example, suppose a heterogeneous system in which one sensor node and one actuator are involved. We will use a PC for configuring the system and ZeroC Ice as object oriented middleware. Figure 3 shows a schematic summary of the sample application.

Suppose a sensor node which measures the level of a tank. When level exceeds a certain limit sensor will active a remote alarm. This value limit may be reconfigured in execution time. Periodically, sensor sends the measured level to an event channel, when others distributed objects (by implementing a concrete interface) may subscribe on it and receive the sensor information.

```
module Dev {
  interface Sensor {
    void setLimit(Byte limit);
  };

  interface SensorOb {
    void update(Byte value);
  };

  interface Alarm {
    void activate();
  };
};
```

The system designer may define the interfaces of the distributed objects in Slice language as shown in previous listing. Sensor observers (`Dev::SensorOb`) will receive the periodic information via `update()` method. In our example, a PC application implements a sensor observer for collecting measurement data and deciding if it is necessary to change the sensor limit value. However, multiple objects may receive this information just subscribing to the event channel.

In the following, assume this code is saved in a file named "tank.ice".

4.1 Alarm actuator node

First one, it being shown how it should be implemented the alarm node using our development model. Alarm node would be seen as "pure server" object. It means alarm object does not perform invocations to other objects.

The driver designer may provide a binary interface similar to the following listing. In picoObject architecture, any event or method are identified by an identifier. Thus, the alarm binary interface is a definition of an only one remote method. In this case, objects that implement the `Dev.Alarm` interface have a method called `activate` identified by the identifier `10`.

```
remote Dev.Alarm activate(10);
```

As pure server, alarm object has not an associated behavior in terms of remote objects. Next listing shows an IcePick description of the distributed scenario, from the viewpoint of the alarm node. This description should be provided by the environment developer.

```
uses "tank.ice";

object Dev.Alarm alarm;

local adapter node {
   endpoint = "xbow -h 20";
   objects  = {"WALARM":alarm};
};
```

In this case, the distributed scenario is quite simple. It is composed by a `Dev.Alarm` object (`alarm`) and a local node in which `alarm` is registered (with the object identifier `WALARM`). This node has a endpoint that use the Xbow radio protocol (for CrossBow devices).

With the Slice interface definition, SIS information and the IcePick description of the distributed scenario, the `ipkc` compiler is ready to generate whole required code to be loaded in the real device.

4.2 Level sensor node

Following listing describes the SIS binary interface. Apart from defined methods in the Slice interface, the driver designer provides `getValue()` and `getLimit()` methods that are sensor locally accessible and should be used for getting level measure and the limit value, respectively.

```
event LIMIT_EXCEEDED(5);
```

```
remote Dev.Sensor setLimit(10);

local Dev.Sensor getValue(20) {
  output = byte;
};
local Dev.Sensor getLimit(21) {
  output = byte;
};
```

The method implementations is hidden to the environment developer. Suppose that the limit value is saved into node when an invocation of `setLimit()` method occurs. Thus, `LIMIT_EXCEEDED` is defined as an asynchronous internal event that occurs when the limit value is exceeded.

With the Slice and SIS information, the environment developer is be able to describe the sensor node scenario and its behavior by using IcePick language. The following IcePick source code is an implementation example for the level sensor node.

```
uses "tank.ice";

object Dev.Sensor sen;
object Dev.SensorOb ob;
object Dev.Alarm alarm;

local adapter sensor_node {
   endpoint = "xbow -h 25";
   objects  = {"WLSENSOR":sensor};
};
remote adapter alarm_node {
   endpoint = "xbow -h 20";
   objects  = {"WALARM":alarm};
};
remote adapter channel_node {
   endpoint = "xbow -h 30";
   objects  = {"WLCHANNEL":ob};
};

timer(60) {
   ob.update(sen.getValue());
}

event LIMIT_EXCEEDED do {
   alarm.activate();
}
```

From sensor viewpoint, the interested objects are the `sensor` itself, a tank level observer `ob` and the `alarm`. The observer `ob` may be a single object, an event channel or any other entity which implements the interface `Dev.SensorOb`.

Now, the `sensor_node` is a local node (target node) and the rest are remote respect to sensor, such as `alarm_node`. Each `adapter` structure has a Xbow endpoint and a set of registered objects by using the `objects` attribute.

After the distributed scenario description, a `timer` trigger is defined which include a sequential set of invocations. In this case, there is only one invocation that is executed every 60 seconds. The measure value is sent to `ob` via `update()` method.

By using `event` trigger, it may be implemented the activation of the alarm. When the internal event occurs(`LIMIT_EXCEEDED`), the alarm will be switched on via `activate()` method.

This is a simple behavior, but IcePick provides triggers for other kinds of events:

- `boot`: associated invocations will be executed at node boot time.
- `when` *INVOCATION* `do`: invocations associated to a `when` trigger will be executed when a local node object receives a certain *INVOCATION*. The trigger invocation parameters are available to be used in the corresponding invocations.

Moreover, IcePick allows to specify conditional invocations that will be executed if the condition success.

4.3 PC node implementation

Middlewares like CORBA or Ice provide tools to translate the interface specification to a concrete implementation language (Java, C++, Python, etc.). In our example is used Slice as interface specification language. So, the application developer may use the tools provided by ZeroC to translate that code.

The application developer does not care about if the server is a sensor node or any other kind of hardware system. The designer just knows the interface which server implements. The following pseudo-code implements how to attach a subscriber into the sensor event channel:

```
class Observer implements NodeOb {
   void update(byte b) {
      saveValue(b);  // process value b
   }
}

[...]

Topic t = resolveObject("xbow -h 30");
Observer ob = new Observer();
t.subscribe(ob);
```

Thus, the observer `ob` will receive the status changes from tank level sensor and it just save that value. Based on received values history, client may change the value limit as follows:

```
byte limit = determineNewLimit(history);
Sensor s = new Sensor("xbow -h 25");
s.setLimit(limit);
```

5. Results

For getting results we have built the ipkc compiler in C++ using Flex [20] and Bison [21] tools for the parsing stage. We have selected Slice as interface specification language. There is a back-end available (written in Python) that generates state machines called picoIce [17]. These state machines can parse IceP [3] messages (ZeroC-Ice protocol). Each state machine represents a node of the distributed system.

Furthermore, we have used different implementations of the FSM virtual machines provided by ARCO research group. The features of these virtual machines are as follows:

- 256 bytes for RAM memory.
- 256 bytes for EEPROM.

- 256 bytes for Flash memory.
- 1 KB for program memory.

Thus, that features limit the generated code complexity. However, this hardware resources are enough for many simple operations like self-state reporting, invoking a remote operation, reactive behavior or even device advertisement.

Table 1 shows the size of some prototypes. Column A shows whether or not the object is able to advertise itself using the ASDF protocol [22], [23]. T is the number of triggers programmed and R/RW means the type of access an object implements (Read only or Read/Write), that is, whether it is possible to change the object state or just to read it. The virtual machine that executes the FSM bytecode requires about 2.2 Kwords and 55 bytes of RAM in an 8-bit MicroChip microcontroller.

Table 1: Result values generated by compiler. Values are expressed in bytes (no mistake).

Objects	A	T#	Data	Code	Flash	Total	RAM
Minimum	-	0	14	89	0	103	18
Bool.R+Bool.RW	-	0	68	374	0	442	19
Bool.RW	X	1	177	322	0	449	19
Bool.R+Active.R	X	2	212	431	100	743	19

"Minimum" is a pure server object, similar to alarm node of the application example shown bellow. The compiler optimize the generated code, so it is possible to create a entirely functional pure server object using 103 bytes of program memory and only 18 bytes for RAM (without virtual machine code).

`Active.R` interface is implemented by objects that report its status change to event channels. Active object should save the references to remote publisher and channel objects.

6. Conclusion

The new proposed development model integrates the SAN programming in the conventional development process of the distributed applications by using standard object oriented middlewares. With ipkc, and the associated language and tools, we can build distributed objects in low-cost, and then constrained-resource, devices. It is a way to create small objects in a conventional general purpose distributed system for applications such as consumer products remote management, ambient intelligence, wireless sensor networks, home networking, etc.

IcePick is a powerful and extensible tool to design distributed scenarios defining the nodes behavior and their interactions. Thus, we propose use the distributed object oriented paradigm for building SAN applications based on the distributed scenarios description and the relations between objects. Any entity of the system may be seen as an object which implements a concrete interface. There is no matter if the object resides in a PC, in a powerful server or resource-constrained node. The main idea of IcePick is that the SAN is not separate from the distributed system; the SAN is part of the system.

Furthermore, compiling IcePick files with ipkc we can test and check the modeled distributed system over a test platform before the final installation in the physical devices.

As future work a plugin to develop hardware versions of the picoObjects as stated in [24] is being designed, so we will be able to design also distributed hardware objects following the same methodology.

Acknowledgements

This research is partly supported by the Ministry of Science and Technology (under grants TEC2008-06553) and by FEDER and the Regional Government of Castilla-La Mancha (under grants PAI08-0234-8083).

References

[1] M. Molla and S. Ahamed, "A survey of middleware for sensor network and challenges," *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pp. 6 pp. –228, 0-0 2006.

[2] *The Common Object Request Broker: Architecture and Specification*, 3rd ed., Object Management Group, 2002. [Online]. Available: http://www.omg.org/spec/CORBA/3.1/

[3] Henning, M. and Spruiell, M, *Distributed Programming with Ice*, Revision 3.3.1, May 2008.

[4] Sun Microsystems Inc., *Java Remote Method Invocation (Java RMI)*, 2006. [Online]. Available: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

[5] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, D. T. W. Kim, B. Zhou, and E. G. Sirer, "On the need for system-level support for ad hoc and sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 2, pp. 1–5, April 2002. [Online]. Available: http://dx.doi.org/10.1145/509526.509528

[6] J. Sung, T. S. Lopez, and D. Kim, "The epc sensor network for rfid and wsn integration infrastructure," *Pervasive Computing and Communications Workshops, IEEE International Conference on*, vol. 0, pp. 618–621, 2007.

[7] TinyDB. TinyDB - A declarative Database for Sensors Networks. [Online]. Available: http://telegraph.cs.berkeley.edu/tinydb/

[8] "Tinyos documentation," 2009. [Online]. Available: http://docs.tinyos.net

[9] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[10] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *NSDI'05: Actas de 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 343–356.

[11] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. Danny, K. Bing, Z. Emin, and G. Sirer, "On the need for system-level support for ad hoc and sensor networks," *Operating System Review*, vol. 36, pp. 1–5, 2002.

[12] A. Boulis, C. C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *MobiSys '03: Actas de 1st international conference on Mobile systems, applications and services*. New York, NY, USA: ACM, 2003, pp. 187–200.

[13] A. Boulis, C. C. Han, R. Shea, and M. B. Srivastava, "Sensorware: Programming sensor networks beyond code update and querying," *Pervasive Mob. Comput.*, vol. 3, no. 4, pp. 386–412, 2007.

[14] C. Srisathapornphat, C. Jaikaeo, and C. C. Shen, "Sensor information networking architecture and applications," *IEEE Personal Communications*, vol. 8, pp. 52 –59, 2001.

[15] S. Li, Y. Lin, S. H. Son, J. A. Stankovic, and Y. Wei, "Event detection services using data service middleware in distributed sensor networks," in *Information Processing in Sensor Networks*. Springer, Jan. 2003, p. 557.

[16] R. Bosman, J. Lukkien, and R. Verhoeven, "An integral approach to programming sensor networks," in *Actas de Consumer Communications and Networking Conference, IEEE*, Jan. 2009. [Online]. Available: http://www.win.tue.nl/san/wsp

[17] F. Moya, D. Villa, F. J. Villanueva, J. Barba, F. Rincón, and J. C. López, "Embedding standard distributed object-oriented middlewares in wireless sensor networks," *Wireless Communications and Mobile Computing*, vol. 9, no. 3, pp. 335–345, 2009.

[18] D. Villa, F. Moya, and J. C. López, "Implementación mínima de objetos CORBA para dispositivos empotrados," in *Actas de XIV Jornadas TELECOM I+D*, 2004.

[19] F. J. Villanueva, D. Villa, F. Moya, F. Rincón, J. Barba, and J. C. López, "Lightweight middleware for seamless hw-sw interoperability, with application to wireless sensor networks," in *Actas de Design, Automation and Test in Europe (DATE)*, Apr. 2007, pp. 1042 –1047.

[20] F. Project, *The Flex manual, version 2.5.35*, 2007. [Online]. Available: http://flex.sourceforge.net/manual/

[21] C. Donnelly and R. Stallman, *Bison: The Yacc-compatible Parser Generator, version 2.4*, 2009. [Online]. Available: http://www.gnu.org/software/bison/manual/

[22] F. J. Villanueva, D. Villa, M. J. Santofimia, F. Moya, and J. C. Lopez, "A framework for advanced home service design and management," *Consumer Electronics, IEEE International Conference on*, pp. 155–156, 2009.

[23] D. Villa, F. Villanueva, F. Moya, F. Rincón, J. Barba, and J. López, "ASDF: an object oriented service discovery framework for wireless sensor networks," *International Journal of Pervasive Computing and Communications (IJPCC)*, pp. 371–389, 2008.

[24] J. Barba, F. Rincón, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. Lopez, "Ooce: Object-oriented communication engine for soc design," *Digital Systems Design, Euromicro Symposium on*, vol. 0, pp. 296–302, 2007.