Lightweight Middleware for Seamless HW-SW Interoperability, with Application to Wireless Sensor Networks

F. J. Villanueva, D. Villa, F. Moya, J. Barba, F. Rincón and J. C. López
Department of Information Technologies and Systems
School of Computer Science
13071, Ciudad Real, Spain.
felixjesus.villanueva@uclm.es

Abstract

HW-SW interoperability by means of standard distributed object middlewares has been proved to be useful in the design of new and challenging applications for ubiquitous computing and ambient intelligence environments. Wireless sensor networks are considered to be essential for the proper deployment of these applications, but they impose new constraints in the design of the corresponding communication infrastructure: low-cost middleware implementations that can fit into tiny wireless devices are needed.

In this paper, a novel approach for the development of pervasive environments based on an ultra low-cost implementation of standard distributed object middlewares (such as CORBA or ICE) is presented. A fully functional prototype supporting full interoperability with ZeroC ICE is described in detail. Available implementations range from the smallest microcontrollers in the market, to the tiniest embedded Java virtual machines, and even a low-end FPGA.

1. Introduction

The perceived value of an ubiquitous system is mainly due to its ability to create and to support end-user services. Firstly we should be able to acquire and distribute as much information from the environment as possible. Then we should be able to build robust scalable distributed applications based on that information. Two major components should interact seamlessly: networked sensor/actuator devices and distributed services.

Wireless sensor networks (WSN) provide an excellent platform for the development of useful ubiquitous computing applications. Our aim is to integrate them in the global system in such a way that software developers do not need to make distinction between interacting applications in a WSN and a typical distributed application in a traditional network.

In this paper, we are mainly concerned with the implementation of minimum cost devices able to support the large variety of device and network technologies currently deployed in the target environments. Besides, we also intend to make them interoperable with traditional distributed software platforms for easier deployment of next generation services.

Our goal, then, is to allow wireless sensor devices to export their capabilities as if they were standard distributed objects. These objects should be able to hide the heterogeneity of underlying technologies such as transport protocols and network architecture and topology.

2. Previous work

There are dozens of distributed object implementations in the market: DCOM, Java RMI, Jini, EJB, CORBA, Web Services, .NET Remoting, ZeroC ICE [2], etc. Most of them rely on a shared networking core, the communication middleware, which provides location and network transparency. Unfortunately, the mentioned implementations of standard object oriented middlewares require too much computing resources for our target devices. In this paper we will center on ZeroC ICE, an excellent CORBA-like middleware, but the same approach is also applicable to the other middlewares.

Lowering the cost of embedding a distributed object middleware has been the focus of some miniaturization efforts. There are three common strategies to reduce the footprint of an embedded middleware [15]: 1) to remove costly features keeping a good degree of genericity, 2) to adapt the middleware to specific devices, or 3) to use proxies.

The first approach is used by the Object Management Group [14] in MinimumCORBA [7], a lightweight version of its widespread CORBA [6] architecture. Others efforts

in this way include e*ORB [1] or dynamicTAO [10] and its descendants: LegORB [16] and UIC-CORBA [18]. Micro-QoSCORBA [9] and nORB [19] also use this first strategy.

A representative of the second approach to the development of small footprint middlewares is PalmORB [17].

The third alternative requires a mediating host to allow interoperability with objects in a standard middleware. This approach is used by UORB [15] and one of the integration alternatives proposed in SENDA [13]. For further details over these three different strategies please see [20].

All these previous works follow the same basic rules: they remove dynamic invocation and dynamic instantiation features, they simplify the interface definition language (OMG IDL in the case of CORBA) removing complex or variable length data types, they also remove some fields from the communication protocol, they also remove or simplify the types of messages used in the protocol, they even drop support for indirect references or common services, and they modularize the communication engine and instantiate only those components which are actually used.

There are also a number of proposals for WSN middlewares such as TinyLime [4], TinyDB [12] or Sensation [8]. Unfortunately they require too much computing resources to be useful in our target devices. Specifically, TinyLime and TinyDB requires TinyOS [12]. Additionallly TiniDB requires a query parser in each node and Sensation provides of a middleware integration architecture by means of a sensor abstraction layer and a XML based unified sensor language.

Besides, all of them introduce specific API's and architectures which contribute to increase the learning curve and make more difficult the interoperability with high level applications (from application developer point of view). We need something much smaller, self-contained, and much cheaper, but with a similar set of features.

3. Minimal footprint

Instead of removing even more features from the middleware, let's think the other way. First we will define the smallest feature set that a distributed object should provide. Then we will consider the overhead introduced for each additional feature and whether they meet the application constraints.

Although it is important that each device looks like a distributed object, it is not essential that they are actual distributed objects. If devices are able to generate coherent replies when they receive predefined request messages then the system will work as expected. For a given communication middleware these requests and reply messages are completely specified by the communication protocol (ICEp in the case of ZeroC ICE or GIOP in the case of CORBA).

If the device is just an application-specific ICEp server it will be seen as a standard object from the network but there is a huge potential for resource savings. The object may get rid of the whole communication engine and its API. There is no need for object adapters, marshalling routines, etc. We just need to implement the message handling code for those messages whose destination is an object placed at the device. Therefore there would be a generated ad-hoc implementation for each device.

We used the above implementation strategy to develop a set of functional prototypes named *picoObjects*. In summary *picoObjects* provide a toolset for the automated generation of code able to replace a standard communication engine in WSNs devices.

Code generation must be performed with careful consideration of the constraints imposed by the target platform. Generated code is obviously different for each platform but it will also differ for servers with a different set of objects, even when the platform and the object interfaces are the same.

It is worth noting that a server implemented using this technique will only reply to messages directed to its resident objects. Messages handled by the communication middleware (such as object location in CORBA) will be silently discarded. It is always possible to include these messages as the methods of a special object if needed.

A communication middleware will usually expose two different interfaces to access every service in the system: 1) at the programming level, it provides a standardized application programming interface which abstracts communication details, protocols, etc.; 2) at the network level it provides a common protocol (ICEp in case of ZeroC ICE) allowing seamless interconnection among communication engines running on different machines.

A *picoObject* lacks in a local communication engine. The server program must include code to perform communication primitives and manage its registered objects. Nonetheless, for the rest of the network a *picoObject* behaves as an usual object. It provides a network level interface without significant differences with respect to a standard object. We may say that a *picoObject* implements a virtual communication engine.

PicoObjects may also handle client-side communications using similar techniques. Client-side messages are composed as a set of templates with just the bare minimum configurable fields.

4. General design of *picoObjects*

The main goal of *picoObjects* is the implementation of the essential features needed for a device to expose a standard object behaviour in the network. From that point we intend to define and develop mechanisms to scale the func-

tionality of the device depending on the constraints imposed by the target platform. Our initial targets range from the smallest and cheapest eight-bit microcontroller (even cheaper and smaller than the common choice for wireless sensor devices) to a standard PC.

It may be argued that generating the message handling code for a whole communication middleware does not offer any particular advantage over a standard middleware. Even in that case there are situations that may favour our approach over full-blown middlewares (reliability, real-time constraints, security, etc.).

We define the minimum set of features using the considerations made in section 2, adding a few additional constraints for the sake of simplicity:

- We always follow the standard message format for the communication protocol.
- We will only support the simplest protocol version whenever interoperability is not compromised.
- We will not support common middleware services (e.g. Naming and Event services).
- Implementations will be fully static.
- Resident objects are *always on*. There is no way to activate or deactivate objects.

This set of features warranties a compliant behaviour with no need of mediator devices. This fact has been verified by means of several prototypes (see section 5).

PicoObjects can be looked up at the middleware ORB (Object Request Broker) and used just like any other object, without any difference for the client.

The simplest way to achieve a coherent behaviour for each *picoObject* is by means of message matching automata. In this context, the allowed message set for a given object constitute a BNF grammar defined by the following elements:

- Message format for the middleware communication protocol.
- Object identity, that is to say, unique object identifiers. It should be noted that several object identities may be backed by a single piece of code. This technique is usually called *default servant* in CORBA parlance.
- Concrete interfaces provided by the object. It includes name, arguments and return value for each method (SLICE declaration in ICE or IDL in CORBA).
- Serialization format (CDR in case of CORBA).
- Standard interfaces inherited from the communication engine (Ice::Object in case of ICE).
- Constraints for the target platform.

From the careful analysis of these elements we define the set of lexical elements (tokens): compulsory fields in each message with a known format and size, object identities and identifiers for methods and interfaces.

Then we generate the rules describing how these tokens may be combined together (the BNF grammar). This information is enough to automatically generate a fully functional parser. The whole development flow is shown in Figure 1.

Every *picoObject* must include a set of user procedures (object method implementations) that must be filled by hand (as in any traditional middleware). When the grammar parser of a *PicoObject* identifies a whole request message, the corresponding user procedure is automatically invoked and a reply message is generated. If the parser fails to identify a valid method request, then the message is discarded and the *picoObject* looks for a new synchronization point at incoming data.

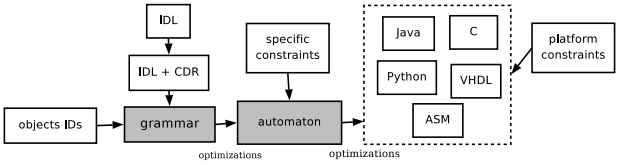

Figure 1. *picoObject* **Development Flow**

Input and output messages may be handled on-the-fly using a generated byte-stream processor. This is a very convenient solution for devices with severe memory constraints (just a few hundred program memory words and a dozen general purpose registers). In this scenario there is no room to even store the incoming message. The request message is processed as the bytes arrive and the reply message is also generated partially from replication of the incoming data. The last part of the reply message is generated by the user procedure for each method.

Our strategy is quite different with respect to previous middleware minimization approaches such as MicroQoSCORBA. MicroQoSCORBA tries coarse grain code minimization by building a custom implementation from predefined libraries. *PicoObjects* use a finer grain code minimization strategy by completely generating the message parsing code for each application.

5. Sample prototypes

The above approach has been applied to a pair of existing middlewares: CORBA and ZeroC ICE, leading to

picoCORBA and picoICE respectively. The constraints imposed by each particular middleware lead to slightly different design decisions. We will describe in depth the ICE prototype as "proof of concept" of a *PicoObject* real implementation.

Both prototypes were developed in Microchip PIC assembler, Java on a standard embedded PC, Java on an embedded Dallas Semiconductors TINI device, C on a standard embedded PC, and VHDL on a Xilinx VirtexE FPGA. We are currently integrating the prototypes in WSNs platforms like Zigbee CC2420 based development kit from Chipcon [3].

The latency for the responses depends on the transport mechanism. Our prototypes using a Lantronix XPort for TCP/UDP encapsulation introduce some latency due to buffering and timeout driven packetization. However, the transport layer may easily be changed without major modifications in the picoObject. For example, the SLIP based prototype begins the transmission of serial response packets well before the end of the request.

Two simple but fully functional prototypes supporting complete interoperability with CORBA and ICE, with specific developing details, are available in our website at [5].

6. PicoICE

ICE is a CORBA-like middlware which implements a feature set unparalleled in any other distributed object platform (object persistence, object migration, authentication, security, replication, deployment services, firewall gateways, etc.). A summary of the differences between ICE and CORBA is available at the ICE home page [2].

ICE already provides a reduced implementation for embedded devices, and offers a few advantages over CORBA to reduce resource usage even further. ICE protocol is simpler than GIOP for a number of design decisions:

- Messages are always little-endian so we do not need to care about byte ordering.
- There is support for unreliable transports such as UDP (much easier to implement in a low cost embedded device).
- There are less types of messages and some of them may not be implemented without compromising interoperability.
- Unprocessed message fields may be easily skipped because they are usually preceded by the field length.
- There are no data alignment requirements for messages on-the-wire.

Embedded middleware	Minimal server code size
TAO	1738 KB + OS
nORB	567 KB + OS
MicroQoSCORBA (TINI)	21 KB + OS
UIC/CORBA	35 KB + OS
ZEN (Java)	53 KB + OS
picoCORBA (Java)	5 KB + OS
picoCORBA (TINI)	4 KB
picoCORBA (C)	5.2 KB + OS
picoICE (C)	5.4 KB + OS
picoCORBA (PIC12C509)	415 words
picoICE (PIC12C509)	503 words

Table 1. Small server sizes on embedded middlewares

The picoICE prototype fully conforms to the ICE protocol specification for connection-oriented transports and connection-less transports. Any reliable or unreliable transport protocol may be used in combination with picoICE objects.

An implementation of a fully operative servant, able to handle method invocations for a set of 64 X10 (well known control protocol used for in-home networking) objects fits on less than 1KB of program memory in a Microchip PIC12f675 microcontroller (see table 1) and needs less than 16 eight bit registers. That is three orders of magnitude smaller than the ZeroC ICE implementation and two orders of magnitude smaller than the ZeroC Embedded ICE implementation.

Currently we support TCP and UDP transports over Ethernet or WiFi through a Lantronix XPort device. A *picoObject* may also be connected to a SLIP (serial line IP) serial port.

In [11] a survey of most used WSN HW platforms and their resource requirements is presented. The required code/data memory for these platforms ranges from 60 KB / 1 KB (MOS) to 128 KB / 64 KB (Btnodes). In Table 1 we can see how our implementation fits in all these platforms.

Although, for the same type of implementation (i.e. C language), picoICE objects are a bit larger than picoCORBA objects this is only due to the extra introspection features supported by ICE, that may be removed if they are not needed.

Any ICE object implements an standard interface called Ice::Object defining a set of common methods. Fortunately some of these methods are already handled by the client stubs provided by the standard middleware interface compiler or by the communication engine at the client side. Therefore there is no need to implement all of them as potential ICEp messages.

The picoICE prototype supports *ice_ping*, *ice_id*, *ice_ids*

and *ice_isA*. These methods add minimal introspection capabilities and the ability to remotely test the existence of an object. This and other interoperability implementation details are explained with detail in [20].

6.1. Asynchronous Advertisements

When a picoICE either needs to communicate a state change or wants to advertise itself to the network it can generate an event using the ICEStorm service (an efficient publish/subscribe standard service for ICE applications). Generating an event in picoICE implementation requires a single ICEp message to a previously known ICE channel.

For example, a minimal advertisement service requires that any *picoObject* sends an event (*adv* operation) containing its own reference (its ICE *proxy* in ICE parlance) to an administrative channel in order to be known by the rest of the network and applications. In order to save memory resources, this first message shares most of the information with the events generated to notify a change in the state of the sensor or actuator.

An ICE packet generated by a *picoObject* event looks like this:

```
Magic Number: 'I','c','e','P'
Protocol: 1,0    Encoding: 1,0
Message Type: Request (0)
Compression Status: Uncompressed (0)
Message Size: 53
Request Message Body
    Request Identifier: 0
    Object Identity Name: publish
    Object Identity Content: asdf
    Facet Name: (empty)
    Operation Name: adv
    Ice::OperationMode: normal (0)
    Invocation Context: (empty)
    Input Parameters Size: 16
    Input Parameters Encoding: 1,0
    Encapsulated parameters (10 bytes)
```

Most of the fields are kept constant for all messages, and only Object Identity name, Object identity content, Operation name and, obviously, the parameters differ.

7. Implications in WSN design flow

Let's see how the design flow for an application over WSN using our approach looks like. At the beginning, it is necessary, like in the design of any distributed application, to specify the object interfaces. The simplest interface that we use for a WSN would be:

```
module iFloat {
  interface R {
    float get();
  };
};
```

In our case, we specify that interface using SLICE, the definition language defined by the ICE platform (CORBA uses IDL and JINI uses JAVA). With this interface we can get, for example, the temperature of a sensor node.

At this point it is necessary to generate an object identity in order to identify the object inside the network: this is called in ICE the universally unique identifier (UUID). The simplest UUID is a string, for example WSN-O1, WSN-O2, etc.

This UUID identifies the object in the network. If a WSN node implements two objects, each object needs its own UUID.

After that, we specify the standard methods of the inherited ICE::Object interface which have to be implemented by the *picoObject* (*ice_ping*, *ice_isA*, ...).

All this information, specified as follows, is gathered in a file that feeds the automatic generator:

```
uses WSN.ice
interface Ice.Object{
    ice_ping,
    ice_isA,
    ice_id,
    ice_ids}
WSN-01: WSN.iFloat.R, Ice.Object
```

Now we need to generate the stubs for the client and the skeleton for the server (the WSN node in our target environment). The client stubs can be generated with the off-the-self SLICE compiler from the interface definition (like in any traditional distributed application). The skeleton for the server is generated with our compiler from the file shown above. The compiler generates a platform independent finite state machine that recognizes the incoming messages and produces the appropriate responses. This finite state machine is then compiled (next step) to several specific languages such as standard C, Microchip PIC assembler, Java or VHDL (for a specific hardware implementation).

In the last step of the design flow, the wireless sensor developer has to write the particular implementation of the get() method for this application, that is, he/she decides the specific actions to be performed when the method is invoked (i.e., to "read" the temperature value from the hardware sensor and then to return the value).

The implementation of the complete client application does not differ from any other traditional distributed application.

As a conclusion the advantages of using our approach to the design of WSN applications include:

- We provide to *the WSN application developer* with generic and high-level programming interface that allows to completely abstract the design from technology-specific WSN issues. The software/hardware objects that resides in the sensor nodes

can be looked up at the communication engine (ORB) and used just like any other object.

- We provide to *the embedded code developer* with a set of tools that automatize the creation of the communication infrastructure that allows a wireless sensor node to interact with the rest of the WSN and even with any other system out of the WSN. The programming and configuration of a sensor network are very difficult and error prone tasks [4]. With our approach, the compiler generates all of the communication related code (obviously, if the platform is conveniently supported). Currently we support the PIC microprocessor family and we are working in the Atmel family (both of them broadly used in the design of WSNs).

- We provide to *the designer of sensor nodes* with a middleware which can be implemented in hardware just specifying a set of basic parameters (the supported interface and the object identifiers). In fact, we have already extended our set of tools to automatize the generation of the VHDL code that corresponds to the hardware version of the communication engine.

8. Conclusions and future research

In this paper we propose a set of techniques for the implementation of distributed objects on low cost WSN devices such as eight-bit microcontrollers. Results show that resource consumption is two orders of magnitude than previously published data on small middlewares implementation.

As they are currently implemented, *picoObjects* exhibit ultra-low latency, since the reply messages are composed on the fly while the object is still receiving the request. This makes them specially suitable for low-latency operation even on low bit-rate networks.

As the basic prototypes still evolve, we are now developing high level tools to deploy a complete *picoObject* network. We are also extending the concept to support other middlewares.

PicoObjects are being used as major components of SENDA, a middleware-based infrastructure for modelling, development, and deploying of next generation ubiquitous computing services [13].

9. Acknowledgements

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).

References

[1] Openfusion e*orb. available online at http://www.prismtechnologies.com/.

[2] Zeroc, inc. *ICE Home Page*. available online at http://www.zeroc.com/.

[3] Chipcon. Cc2420 zigbee dk user manual v1.0. Available online at http://www.chipcon.com.

[4] C. Curino, M. Giani, M. Giorgetta, and A. Giusti. *TinyLIME: Briding Mobile and Sensor Networks through Middleware*. *PerCom*, 2005. Hawaii. USA.

[5] A. Group. Picoobject web demostration example. http://arco.inf-cr.uclm.es/marisa.html.en.

[6] O. M. Group. *The Common Object Request Broker: Architecture and Specification*. Technical report, Object Management Group, 1999. Available in http://www.omg.org/.

[7] O. M. Group. *Minimum CORBA Specification*. Technical report, Object Management Group, August 2002. available online at http://www.omg.org/.

[8] T. Hasiotis, G. Alyfantis, V. Tsetsos, O. Sekkas, and S. Hadjiefthymiades. *Sensation: A middleware Integration Platform for Pervasive Applications in Wireless Sensor Networks*. *EWSN*, 2005. Instambul Turkey.

[9] Haugan and Olav. *Configuration and Code Generation Tools for Middleware Targeting Small Embedded Devices*. Master's thesis, M.S. Thesis, Dec 2001.

[10] F. Kon, F. Costa, G. Blair, and R. Campbell. *The Case for Reflective Middleware*.

[11] M. Kuorilehto, M. Hannikainen, and T. Hamalainen. *A survey of Application Distribution in Wireless Sensor Networks*. *EURASIP Journal on Wireless Communications and Networking*, pages 774–788, 2005.

[12] S. Madden, J. Hellerstein, and W. Hong. *TinyDB: In-Network Query Processing in TinyOS*, September 2003. Version 0.4.

[13] F. Moya and J. López. *SENDA: an alternative to OSGi for large-scale domotics*. *The Proceedings of the Joint International Conference on Wireless LANs and Home Networks (ICWLHN 2002) and Networking (ICN 2002)*, 2002. World Scientific Publishing, pp 165-176.

[14] OMG. Object management group. http://www.omg.org.

[15] Rodrigues, G., Ferraz, and C. *A CORBA-Based Surrogate Model on IP Networks*, 2001.

[16] M. Roman, M. Dennis, Mickunas, F. Kon, and R. Campell. *LegORB and Ubiquitous CORBA*. Technical report, Feb 2000.

[17] M. Roman and A. Singhai. *Integrating PDAs into Distributed Systems: 2K and PalmORB*. HUC, 1999.

[18] M. Román, F. Kon, and R. H. Campbell. *Reflective Middleware: From Your Desk to Your Hand*, 2001.

[19] V. Subramonian and G. Xiang. *Middleware Specification for Memory-Constrained Networked Embedded Systems*, 2003.

[20] D. Villa, F. J. Villanueva, F.Moya, F. Rincón, J. Barba, and J. C. López. Embedding a general purpose middleware for seamless interoperability of networked hardware and software components. *In proceedings of Grid And Pervasive Computing (GPC), LNCS, Taiwan*, 2006.