Embedding a Middleware for Networked Hardware and Software Objects*

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

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

Abstract. In this paper we present a novel approach to the design of ubiquitous computing environments based on an ultra low-cost implementation of standard distributed object middlewares suitable for networked hardware and software components of the system.
We prove the feasibility of our approach with a set of prototypes supporting basic interoperability with CORBA and ZeroC ICE. In some cases, the resulting embedded prototypes are two orders of magnitude smaller than previous implementations of small objects. They are suitable for embedding into the smallest microcontrollers in the market, or in the tiniest embedded Java virtual machines, or even in a low-end FPGA.

1 Introduction

A useful ubiquitous computing environment must be able to perceive stimuli from the physical world and react on them. The perceived value of an ubiquitous system is mainly due to its ability to create and to support end-user services, based on information from the environment. In this paper we face the problem of developing effective communication mechanisms among a large set of heterogeneous devices including, but not limited to, desktop computers, embedded computers, small microcontrollers, customized FPGA devices, etc. 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.

Our approach departs from many previous heterogenous device network architectures by requiring each device to be autonomous, in the sense that our devices and basic services will work even when all available service gateways fail. We believe this is the easiest way to achieve better robustness, reliability and fault tolerance at a minimum cost. Intermediate elements such as the gateways advocated by e.g. OMG Smart Transducers [3] or OSGi [6] should be avoided in most applications. It should be possible to implement autonomous

* This research is supported by FEDER and JCCM, under Grant PBC-05-009-1, and by Spanish Ministry of Education, under Grant TIN2005-08719

services whose correct operation does not depend on the correct operation of any gateway.

Therefore our main goal is to allow embedded devices to offer their capabilities as standard distributed objects. These objects should be able to hide the heterogeneity of underlying technologies such as transport protocols and network architecture.

The seamless communication of heterogeneous distributed components is usually approached in software environments by using a unifying element, the communication middleware. Unfortunately, current implementations of standard object oriented middlewares (DCOM, Java RMI, Jini, EJB, CORBA, Web Services, .NET Remoting, ZeroC ICE [26], etc.) require too much computing resources for many of our target devices.

2 Related work

Many previous initiatives have been oriented towards the miniaturization of existing middlewares. Indeed, the Object Management Group [1] published the MinimumCORBA specification [8], a lightweight version of its popular CORBA architecture [2]. MinimumCORBA removes the most expensive features of the communication engine keeping a good degree of interoperability with standard CORBA objects.

As stated in [17] there are three main approaches to the minimization of distributed object implementations: 1) Remove costly features but keep genericity, 2) adapt the middleware to specific devices, 3) use proxies.

The first approach is used in dynamicTAO [10] and its descendants: LegORB [12] and UIC-CORBA [9]. LegORB is a modularized ORB with the ability to be dynamically configured. The monolithic library of TAO [7] is decomposed in a set of independent functional components that may be omitted from the target application. It is reported that a client-only CORBA application under 20 KB may be built on a HP Jornada 680 running Windows CE, and a 6 KB client-only may be built on a PalmPilot running PalmOS 3.0 (see [13]).

UIC (*Universally Interoperable Core*) define a component based middleware skeleton. Each component encapsulate a small set of features and may be dynamically loaded depending on the running platform, device and network used. UIC, as its name states, may be used to implement communication engines for different middlewares besides CORBA, such as Java RMI or DCOM. A CORBA static server is reported to be 35 KB on a SH3 running Windows CE.

A similar commercial product is e*ORB [14], a modular communication engine with real-time features able to run on a HP iPAQ or a Texas Instruments TMS320C64X DSP.

Another representative of the first approach to the development of small communication engines is MicroQoSCORBA [11]. A customized communication engine may be generated from a set of predefined pieces in order to implement servers and clients suited to a specific application and device (it has been tested on SaJe [24] and TINI [23]).

nORB [15] implements a set of pluggable transport protocols, including some environment-specific protocols (ESIOP in CORBA terminology). It borrows many ideas from MicroQoSCORBA, such as the simplified version of the GIOP standard protocol, called GIOPLite.

A representative of the second approach to the development of small footprint middlewares is TINIORB [18], a MinimumCORBA communication engine customized for the TINI device from Dallas Semiconductor. PalmORB [19] is another example of this approach.

The third alternative requires a mediating host to allow interoperability with objects in a standard middleware. This is the approach used in UORB [17] and one of the integration alternatives proposed in SENDA [21].

Another interesting proposal of the same type is [22]. This work shows how a set of small 8 bit microcontrollers may be published as a set of CORBA objects. The host runs a proxy object for each connected device and communications between each device and a the mediating host use a specialized protocol.

All these previous works follow the same basic rules: Remove dynamic invocation and dynamic instantiation features, simplify the interface definition language (OMG IDL in the case of CORBA) removing complex or variable length data types, remove some fields from the communication protocol, remove or simplify the types of messages used in the protocol, do not support indirect references, do not support common services, modularize the communication engine and instantiate only those components that are actually used.

It should be noted that the above mentioned communication engines require a lot of support utilities: data type marshalling, communication primitives, operating system, etc. Therefore, the actual resource requirements may be orders of magnitude larger than cited.

Even the smallest of the previous distributed object implementations is much larger than feasible on our target environment. Requiring a TINI (around 30 euro) for each device in the ubiquitous system would lead to astronomical prices for useful systems. Just thinking of a RMI-enabled Java virtual machine for each bulb or switch in a building is reserved to millionaires.

We need something much smaller, self-contained, and specially much cheaper, but with a similar set of features.

3 The smallest object

Instead of reducing the features provided by the middleware even more, let's think the other way. We will define the smallest implementation of a distributed object. From that point we will consider the overhead introduced for each additional feature when the application constraints allow them.

From the perspective of the ubiquitous system it is important that each device looks like a distributed object. But 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 request an reply messages are completely specified by the communication protocol (GIOP in the case of CORBA).

If the device is just an application-specific GIOP server it will be seen as a normal object from the rest of the network but there is a huge advantage 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 we propose a generated ad-hoc implementation for each device.

In this paper we proposed PicoObjects as a materialization of the above implementation strategy. In summary PicoObjects provide a toolset for the automated generation of code able to replace a standard communication engine in low-end computing resources.

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 interfaces of the objects are the same.

It is worth to note 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: At a programming level it provides a standardized application programming interface. It abstracts communication details, protocols, etc. At a network level it provides a common protocol (GIOP in the case of CORBA) allowing seamless communication among communication engines running on different machines.

A picoObject lacks 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 significative differences with respect to a standard object. We may say that a picoObject implements a virtual communication engine.

Although it is already implicit in the context, it is worthy to note that a picoObject implement only the server-side of the communication middleware. This is consistent with the idea of developing remote interfaces for each device. The devices behave as small servers.

4 Functionality scaling

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 this point we intend to define and develop mechanisms to scale the functionality

of the device depending on the constraints imposed by the target platform. Our initial targets range from an eight bit microcontroller to a standard PC.

Although the proposed model allows an implementation at almost any conceivable scale, our main targets were the smallest available computing devices. It may be argued that generating the message handling code for a whole communication middleware do not offer any particular advantage over a stantard middleware. Even in this case there may be constraints in the target system that make our approach more advisable (reliability, real-time constraints, security, etc.).

We define the minimum set of features using the considerations of section 2, adding a few additional constraints: a) On one hand we always follow the standard message format for the communication protocol. Using modified protocols (such as GIOPLite in the case of MicroQoSCORBA) implies the need for a mediating element (bridge) responsible for the transformation of messages to allow seamless interoperability. This would contradict our intention to make devices immediately available on the network. b) We will only support the simplest protocol version whenever interoperability is not compromised. c) Resident objects are *always on*. There is no way to activate or deactivate objects.

5 A strategy for small objects

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: a) The message format for the middleware communication protocol. b) The object identity, that is to say, 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. c) Concrete interfaces or set of interfaces provided by the object. It includes name, arguments and return value for each method. d) The marshalling procedure (CDR in case of CORBA). e) Standard interfaces inherited from the communication engine (CORBA::Object in case of CORBA). And f) Constraints of the target platform.

We first define the set of lexical elements (tokens): compulsory fields in each message with a known format and size, object names, method names, interface names. Then we generate the rules describing how these tokens may be combined together (the BNF grammar). This information is enough to automatically generate a complete 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 syncronization point.

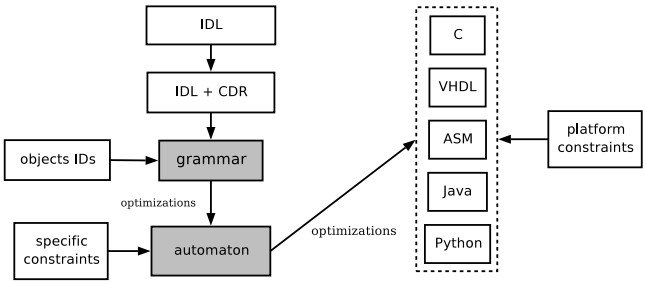

Fig. 1. Development flow of a picoObject

Input and output messages may be handled on-the-fly using a custom byte-stream processor. This is a very convenient solution for devices with severe memory constraints (just a few hundreds of program memory words and a dozen of 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.

In order to lower the memory requirements for token parsing we reduce them using a digital signature, a CRC code or just a checksum. Therefore, even when tokens may be arbitrarily long, the picoObject compiler substitutes it by a length and a single byte checksum. When the picoObject is parsing a request it may just incrementally calculate the input message checksum and check it when the length matches. Actually we do not need to check at every token boundary whether calculated and stored checksums match. If we arrange the set of implemented messages for a given object as a lexical tree then we just need to check at every forking point in order to decide the branch to follow.

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.

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. As illustration of the applicability of this work we will summarize in the following sections the features and design decisions of each prototype.

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 Virtex E FPGA.

6 PicoCORBA

CORBA is now a mature distributed object architecture and a lot of effort has been devoted to embedded CORBA implementations. Most of this previous work is influenced by MinimumCORBA, a reduced footprint specification which removes complex CORBA features keeping a good degree of interoperability with standard CORBA. MinimumCORBA objects are completely standard compliant and they may also be built on full CORBA engines. PicoCORBA goes much further with respect to removing features. PicoCORBA objects are not portable at all since they are usually implemented using a specific assembler language. Even if we use C or any other low level programming language there is no enforcement of any standard mapping since there is no need to link against a common library. The picoCORBA prototype is able to parse a byte stream coming from the network and generate a response. The transport protocol may range from TCP over Ethernet, through SLIP, SNAP, LonTalk, or any other reliable transport protocol.

As described above, there are two key points in which we should check the calculated checksum against the expected checksum: when we receive the object identity and when we must choose among the implemented methods. In order to simplify this procedure even further, we assume that the length of the identity string (`object_key` field) of every picoObject is exactly the same. This assumption do not introduce interoperability problems at all. Object identities will appear in the generated object references and clients are required to use it without modifications when sending requests.

CORBA standard mandates the implementation of GIOP communication protocol to ensure interoperability across the network. PicoCORBA is currenly GIOP 1.0 conformant. This does not introduce interoperability problems since the CORBA standard dictates that any updated GIOP protocol must be backwards compatible.

GIOP dictates that peers which initiate a connection determine the byte order used. With GIOP 1.0 the client is always the initiator and therefore the server is required to adapt to the requested byte order. PicoObjects are supposed to stay in a controlled environment and therefore implementing a single byte order may be acceptable. If this simplification cannot be afforded then picoCORBA objects must implement little endian and big endian versions of all the messages, virtually doubling the resources needed.

Any CORBA object implements a standard interface called CORBA::Object defining a set of common methods. Fortunately some of these methods are already handled by the remote proxy or by the communication engine at the client side. Therefore there is no need to implement all of them as possible GIOP messages. We identified the bare minimum set of common methods to `non_existent` and `is_a`. The former allows the client to know whether the object is willing to answer requests. The latter offers minimal introspection capabilities. Both of them are implicitly implemented in every generated picoObject even when no explicitly stated.

Embedded middleware	Minimal server
TAO	1738 KB
nORB	567 KB
UIC/CORBA	35 KB
JacORB (Java)	243 KB
ZEN (Java)	53 KB
MicroQoSCORBA (TINI)	21 KB
picoCORBA (C)	7 KB
picoCORBA (Java)	5 KB
picoCORBA (TINI)	4 KB
picoCORBA (PIC12C509)	415 words

Table 1. Size of a small server on embedded middlewares

A few limitations apply to the set of implemented messages. Experimental results show that interoperability is not compromised against all tested implementations of CORBA (TAO, OmniORB, MICO, JacORB, JDK1.4, ORBit2). We ignore the contents of the field `requesting_principal` for every incoming message which is already deprecated. Reply messages reproduce two fields from their matching request messages: `service_context` which encapsulates engine specific data and `request_id` which matches requests and replies. Field `reply_status` always contains `NO_EXCEPTION` since picoCORBA does not currently support exceptions or indirect proxies (location forward). We ignore *cancel request* messages. This is explicitly allowed by the CORBA specification. PicoCORBA does not implement *Locate request* or *Close connection* messages. Locate requests may be used by the client to optimize bandwidth when using indirect proxies. PicoCORBA objects are "always on". Therefore there is no need to ever generate *Close connection* messages. Finally PicoCORBA ignores any unhandled message. In particular it ignores any malformed messages and error reporting messages.

An implementation of a fully operative servant, able to handle method invocations for a set of 64 X10 objects fits on 415 program memory words of a Microchip PIC12f675 and requires less than 16 eight-bit registers. That is two orders of magnitude smaller than any other previous implementation of small embedded middlewares (see table 1).

7 PicoICE

ZeroC, Inc. developed a high quality distributed object framework called ICE (Internet Communication Engine) built upon the experience of CORBA but free of legacy or bureaucracy constraints. It 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 [26].

Despite the current lack of support for embedded platforms, ICE offers a few advantages over CORBA to reduce resource comsumption even further. ICE protocol is simpler than GIOP for a number of design decisions: 1) messages are always little endian so we do not need to care about byte ordering, 2) there is support for unreliable transports such as UDP (much easier to implement in a low cost embedded device), 3) there are less types of messages and some of them may not be implemented without compromising interoperability, 4) unprocessed message fields may easily be skipped because they are usually preceded by the field total length, 5) there are no data alignment requirements for messages on-the-wire.

The picoICE prototype is fully conformant with the ICE protocol specification for connection-oriented transports and connection-less transports. Any reliable or unreliable transport protocol may be used in combination of picoICE objects. An implementation of a fully operative servant, able to handle method invocations for a set of 64 objects fits on 478 words of program memory in a Microchip PIC12f675 microcontroller 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 SLIP (serial line IP) serial port.

As in the case of CORBA, ICE requires that every object implement a set of common methods. 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. These features may be removed if not needed.

8 Conclusions and future research

In this paper we propose an alternative implementation of distributed objects for low cost embedded devices such as eight bit microcontrollers or FPGAs. 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 real-time operation even on low bit-rate networks. Exact figures of latency depend on the transport protocol used, which is currently independent of the picoObjects.

As the basic prototypes still evolve, we are now developing high level tools to deploy a 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 modeling, development, and deploying of next generation home services [21].

References

1. OMG (*Object Management Group*), http://www.omg.org/
2. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, ed. 2.3, June 1999. Available in http://www.omg.org/, document id: 98-12-01.
3. Object Management Group, *Smart Transducers Interface Specification*, ed. 1.0, January 2003. Available in http://www.omg.org/, document id: 03-01-01.
4. OMG, *General Inter-ORB Protocol 2.3*, Available in http://www.omg.org/ (Document id: 98-12-01), June 1999.
5. Sun Microsystems, *Jini Architecture Specification*, ed. 1.2, available online at http://www.sun.com/,
6. Open Services Gateway Initiative, *OSGi Service Platform*, ed. 2.0, October 2001, available online at http://www.osgi.org/.
7. The ACE ORB, available online at http://www.theaceorb.com/.
8. Object Management Group, *Minimum CORBA Specification*, ed. 2.3, August 2002, available online at http://www.omg.org/, document id: 02-08-01
9. M. Román, Fabio Kon, Roy H. Campbell, *Reflective Middleware: From Your Desk to Your Hand*, 2001.
10. Fabio Kon, F. Costa, G. Blair, Roy Campbell. *The Case for Reflective Middleware.*
11. Haugan, Olav. *Configuration and Code Generation Tools for Middleware Targeting Small, Embedded Devices*, M.S. Thesis, Dec 2001.
12. Manuel Roman, M. Dennis, Mickunas, Fabio Kon and Roy Campell. *LegORB and Ubiquitous CORBA*, Feb 2000.
13. LegORB, available online at http://choices.cs.uiuc.edu/2k/LegORB/.
14. OpenFusion e*ORB, available online at http://www.prismtechnologies.com/.
15. V. Subramonian, G. Xiang. *Middleware Specification for Memory-Constrained Networked Embedded Systems*, 2003
16. C. Gill, V. Subramonian. *ORB Middleware Evolution for Networked Embedded Systems*, 2003
17. Rodrigues, G., Ferraz, C., *A CORBA-Based Surrogate Model on IP Networks*, 2001.
18. J. Morena, F. Moya, J.C. López. *Implementación de un ORB para Dispositivos Empotrados*, Sep 2002.
19. M. Roman, A. Singhai, *Integrating PDAs into Distributed Systems: 2K and PalmORB*, HUC 1999.
20. M. Connolly, *CORBA Middleware for a Palm Operating System*, Sep 2001.
21. F. Moya, J.C. López. *SENDA: an alternative to OSGi for large-scale domotics*, Networks, The Proceedings of the Joint International Conference on Wireless LANs and Home Networks (ICWLHN 2002) and Networking (ICN 2002), World Scientific Publishing, pp 165-176, Aug, 2002.
22. W. Nagel, N. Anderson. *A Protocol for Representing Individual Hardware Devices as Objects in a CORBA Networt*, July 2002.
23. Tiny Internet Interface. Available online at http://www.ibutton.com/TINI/index.html
24. SaJe, Real-Time Native Java Execution. Available online at http://saje.systronix.com/.
25. E. Gamma, R.H., R. Johnson, J. Vlissides, *Design Pattens, Elements of Object-Oriented Software*. 1995, Addison-Wesley.
26. ZeroC, Inc., *ICE Home Page*, available online at http://www.zeroc.com/,