Embedding standard distributed object-oriented middlewares in wireless sensor networks[†]

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

28 August 2007

Abstract

This article provides an analysis of the design space available to middleware developers in the context of wireless sensor networks. We identify the weaknesses of current communication abstraction layers and propose alternative implementation techniques that preserve most of the useful features but minimizes the implementation cost in resource constrained wireless sensor nodes. Our proposal includes a whole WSN development framework based on standard distributed objects and a set of specific services designed to support highly dynamic and scalable WSN applications.

distributed objects; sensor middleware; device interoperability

1 Introduction

Wireless sensor networks constitute a specialized subfield of distributed heterogeneous computing where resource and power consumption are major issues. Besides, the dynamic nature of WSN systems poses new challenges for the application developers.

Since early 80s the research community produced a large quantity of abstractions aimed at reducing the inherent complexity of distributed application development. Today most distributed applications are built on top of an intermediate layer of software, known as communications middleware, which defines protocols, a set of basic services and a programming interface to be used by application developers. One of the most successful programming models today is the distributed object paradigm which tries to extend the semantics of standard objects to allow remote method invocations. Examples of this type of platforms are CORBA, ZeroC ICE, and Jini. Most of them also allow application developers to use an heterogeneous mixture of programming languages, development tools, runtime environments and hardware platforms.

Some previous works devised strategies for embedding an ad-hoc middleware into low-cost devices but little effort is made to guarantee interoperability with

[2]This work was supported by FEDER and JCCM under grant PBC-05-0009 and by the Spanish Ministry of Education under grant TIN2005-08719.

standard platforms and compatibility with standard tools. The main motivation of this work is providing an infrastructure that allows developing applications on wireless sensor networks (WSN) using the same methodology that is currently used with standard distributed OO middlewares.

Typical research platforms used in wireless sensor nodes use oversized resources for many applications. For example, external memory ranges from 32kB for WesC platforms to 64 MB for the WINS3.0 platform [1, 14]. Similarly, the ATmega128L microcontroller included in the most widespread platform, the MICA family, is more than an order of magnitude more expensive than the smallest microcontrollers in the market. An illustration of this point can be found in almost any ambient intelligence applications which may require deployment of hundreds of small devices such as switches, motion sensors, dimmers, proximity sensors, etc. Moreover, these platforms are insufficient when applications demand high data rates such as video processing.

From the application developers point of view, a wireless sensor or actuator node may behave as a standard distributed object although the internal implementation may differ considerably. We prove the feasibility of our approach with a set of prototypes which support basic interoperability with CORBA and ICE. We will provide a detailed analysis of the tradeoffs involved in each target platform stressing the conceptual improvement over current practice. Moreover, we describe a set of common services built on standard distributed objects designed for wireless sensor/actuator networks, including a service discovery framework and event propagation for high data rate applications such as multimedia streaming or environment monitoring (e.g. telemetry).

This article is organized as follows. First, we briefly describe a broad range of related research in order to identify the open issues which motivate our work. Afterwards we discuss the tradeoffs involved and the overall application development flow. Then we will describe how these tradeoffs translate into specific design decissions in the prototypes. Finaly we will draw some conclusions and outline future research work.

2 Related work

In this section we will describe previous research on communication middleware platforms for wireless sensors and also some previous efforts aimed at the reduction of the memory footprint of standard distributed objects. Finally we will identify open issues not currently addressed.

2.1 Middlewares on wireless sensor networks

From the point of view of the programming model most wireless sensor network research platforms may be grouped into three categories [1, 19, 14]: TinyLIME [3], TinyDB [6], or Cougar [15] provide a **distributed database** abstraction to access the wireless sensor network; SensorWare [16, 17] or Impala [18] use **agent-based** abstractions where small independent pieces of code are able to migrate through the nodes which implement an interpreter for the agent code; DSWare [20] or TinyDiffusion [4] use an **event-based** approach notifying asynchronously changes in any magnitude to all interested parties. Unfortunately all of these previous works define an ad-hoc infrastructure to develop WSN applications. Interaction with standard distributed objects must be implemented as ad-hoc bridges or adapters.

The integration of heterogeneous WSN technologies has been addressed before in Sensation [2] by means of a sensor abstraction layer and a common XML syntax to describe all sensors. Sensation aims at providing a high level API to manage different WSN through request-response proxies and technology-specific drivers embedded in the sensor abstraction layer. This approach manages heterogeneity of WSN but again it does not handle the interaction with components outside the WSN environments.

2.2 Reducing the footprint of distributed objects

Many previous initiatives have been oriented towards the miniaturization of existing distributed OO middlewares by removing costly features, and intense modularization, but keeping genericity. This is used in dynamicTAO [24] and its descendants LegORB [25] and UIC-CORBA [23]. It is also the approach of MicroQoSCORBA [29], e*ORB [26] and nORB [27].

UORB [30], and the embedding strategy described in [28] and also one of the integration alternatives proposed in SENDA [7] use a different approach based on intermediate proxies running in a mediating host. This may lead to large resource savings at the cost of requiring specialized device protocols and a specific interoperability middleware.

All these previous works share the same basic techniques: dynamic invocation and dynamic instantiation features are removed, the interface definition language (OMG IDL in the case of CORBA) is simplified by removing complex or variable length data types, optional fields in messages are ignored, optional protocol features are either simplified or removed entirely (e.g. error reporting messages), indirect references references are not supported, common services are not supported, the communication engine follows a modular design and applications instantiate only those components that are actually used.

Unfortunately all these previous works leave some open issues that should be addressed by a middleware for wireless sensor networks: (1) traditional WSN middlewares and previous implementations of small distributed objects require relatively expensive platforms; (2) the integration of WSN usually requires complex gateways which handle protocol conversion issues, type-system conversion, API adaptation, etc; (3) existing solutions introduce a new design-flow which requires learning new protocols, APIs and even new languages; (4) existing middlewares for WSN lack industrial support for highly scalable services.

We need something much smaller, self-contained, and capable of being embedded in cheaper platforms. But at the same time it should allow easy integration of sensors without complex gateways or interfaces and minimizing the learning curve for application developers. Besides it should be able to take advantage of existing well-known standard technologies.

3 Embedding distributed objects

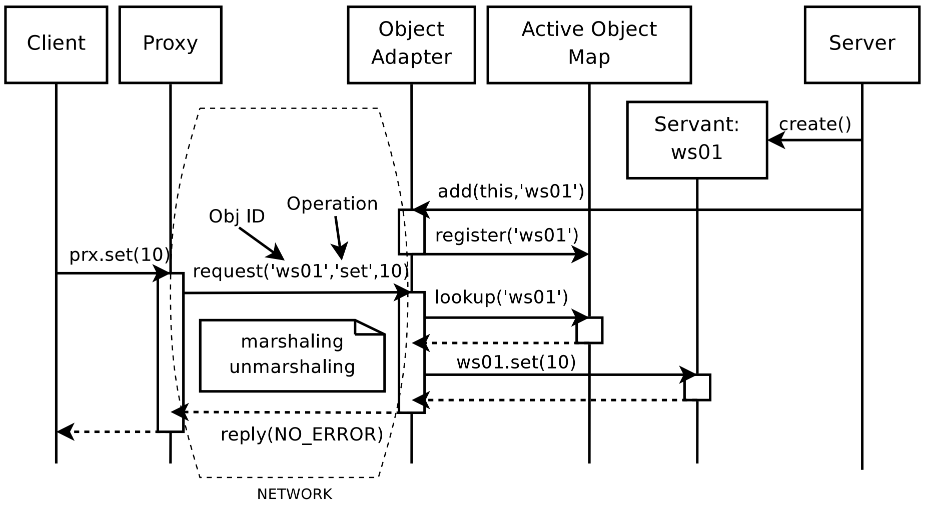

Figure 1: Simplified sequence diagram for standard RMI interaction.

In figure 1 we show a simplified diagram for a remote method invocation (RMI) using a middleware such as CORBA. In a conventional CORBA communications engine (object request broker in CORBA parlance), a remote invocation involves the cooperation of a large number of entities (one or more *object adapters*, a set of *servants* which implement the objects behavior, and an *active object map* which keeps track of object-servant pairs).

Each remote method invocation translates into a request including three important components: (1) an *object id*, which uniquely identifies the target object in the whole network, (2) the *operation*, that determines what member function in the servant must be executed, and (3) the set of *arguments* required by the operation.

Any interaction between a client and an object is achieved exclusively by message passing using standardized protocolos such as the *General Inter-ORB Protocol* (GIOP) in CORBA, or ICE protocol (IceP) in ZeroC ICE. Therefore any implementation of an object remains compatible as long as it conforms to the same message exchange pattern.

In this paper we propose a simplification of the standard RMI implementation suitable for very low cost embedded objects (*picoObjects*), which emulate the server-side message parsing and reply message generation by means of an automatically generated finite state machine. Between this approach and the standard behavior of figure 1 there is a whole range of variants. What is nice about all these implementations is that the development flow is almost unchanged. They all start from the description of the remote interfaces exposed by the objects. Then a stub compiler is used to automatically generate both client-side and server-side skeleton code for each remote interface. By contrast a picoObject compiler would merge all the remote interfaces exposed by a single server into an ad-hoc finite state machine able to parse invocations of any of the available remote methods. In section 4.4 we will describe in detail the implementation techniques used in our compiler prototype. In the following section we will discuss the tradeoffs involved in the emulation of the remote method invocation message interchange pattern. We will use ZeroC ICE as the reference middleware for the sake of simplicity although most of the discussion may be extended to other distributed object implementations.

3.1 Implementation tradeoffs

Implementations of embedded distributed objects must balance the resources required for each wireless sensor node and the features available to the application developer. For example, a reply-message replicates the same request identifier included in the matching invocation message. This is useful for duplicate filtering and out-of-order method dispatching but depending on the sensor node resources we may keep these features to a bare minimum. Likewise most middlewares provide a range of optional encodings including compression, encryption and different data formats. Any of these features may be discarded in order to save some implementation cost.

The same message exchange pattern may be implemented using a large variety of technologies depending on the specific needs. There is a multi-dimensional design space defined by physical magnitudes such as power consumption, time required to perform a remote invocation, bandwidth efficiency, required computing resources, etc. Besides there are some functional aspects of the middleware that may also be analysed as a continuum.

For example, let us consider the dynamic nature of object creation and destruction. A conventional middleware relies on the constructor/destructor support from the underlying implementation language to provide dynamic creation and destruction of distributed objects. But sometimes this is not needed at all in a wireless sensor. A motion sensor, a temperature sensor or a proximity sensor are all physical devices exposed to the network through a set of predefined protocols. This is a completely static picture that could be implemented much more efficiently with a finite state machine (FSM) able to parse the right request messages. Note that this is not object-oriented at all since the FSM must be able to handle requests for all the objects defined in a given node. There is a lot of coupling but this may be handled automatically by a compiler.

Of course there are design choices between a fully static approach and a fully dynamic approach. We may use a memory to store the FSM in order to allow a complete reconfiguration. Or we may use hierarchical finite state machines in order to allow partial reconfiguration. We are still using a non object-oriented approach but it would be possible to simulate the construction/destruction of objects from the point of view of the remote clients.

The efficiency of the communication protocol is another functional axis that we should care of. Distributed object interaction is assumed to be slow. Two orders of magnitude slower than standard method calls. This is mostly due to argument serialization. Lots of extra instructions are required to read data from memory and send them to the destination object while a standard method call may just pass a memory reference. But this is not an issue for a hardware implementation of the objects. Hardware modules already serialize data to interact with other modules and a distributed object is anything that complies with the message interchange pattern described above, even if it is a FPGA.

Besides, hardware objects and on-chip buses allow a broader range of implementation strategies. For example, pipelining may be used to boost the throughput. The generation of the reply message may start even before the whole request message arrived.

This is not just an optimization but a fundamental conceptual change. All current distributed object implementations assume that remote interfaces must be kept to a minimum and designed with careful consideration of the incurred overhead. But in hardware this is no longer true. Remote invocations are essentially free and therefore the middleware may be used for module composition and also as a hardware modeling aid. Nonetheless we will focus on the hardware implementation as the choice for highly demanding sensor nodes. A detailed discussion of this implementation alternative is out of the scope of this article.

A third functional axis to be considered in the middleware implementation deals with the complexity of the required gateways for interconnection of different network technologies. When protocols are similar and the object models match then the gateway implementation is reduced to simple header rewriting. Unfortunately the long experience of these integration efforts show that when the object models do not match then gateways face all sorts of shortcomings (e.g. CORBA-DCOM bridge). When the middleware is being embedded in a controlled environment there are even more options to consider. For example, a hardware implementation would benefit from the adaptation of the addressing mechanism to the internal bus protocols carefully keeping the same type-system and serialization rules. Interaction between two hardware modules may be made very efficient at the cost of an extra addressing scheme adaptation logic in the gateways required for off-chip interaction.

3.2 WSN application design flow

As long as a well-established middleware is directly supported by a set of wireless sensors and actuators there is nothing special in the design flow used by the WSN application developer. Most services and end-user applications are developed using a conventional flow and any of the commercially available tools.

However there are small differences in the design flow used in sensors and actuators (see figure 2). A single sensor node may be able to implement different

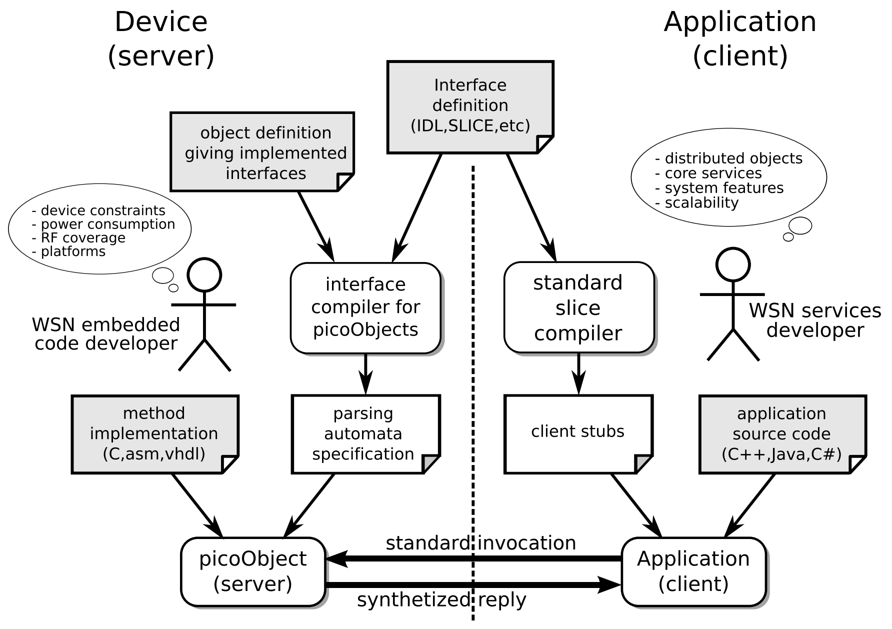

Figure 2: PicoObject design-flow

remote objects but probably there is no need for dynamic object creation and destruction. Therefore we may use an ad-hoc finite state machine (picoObject). The FSM is automatically generated by a compiler from a specification of the set of messages to be parsed. This is easily achieved by means of a standard interface specification and an additional file declaring the whole set of objects to be implemented.

Although we just need to specify the application-level interface many middlewares assume an implicit interface implemented by any distributed object. For example, ICE assumes an implicit *Object* interface with support for minimal introspection. The compiler must be able to generate code to parse those implicit methods as well. In severely constrained environments we may instruct the compiler to generate just the bare minimum.

Client-side stubs can be generated using an off-the-self Slice compiler from the interface definition and the server-side skeleton is generated using the picoObject compiler. The compiler generates a platform independent specification of the finite state machine which is then translated into a specific implementation language. Standard C, Microchip PIC assembler, Java and VHDL are currently supported.

The final step of the development flow should fill the generated skeletons with functional code. The developer must write by hand the specific functionality of each method as in any traditional object oriented middleware.

4 Interoperability of embedded objects

The tradeoffs identified above must be carefully considered on a case-by-case basis when embedding any commercial middleware. In this section we will show how specific constraints of two particular distributed middlewares translate into efficient embedded implementations for wireless sensor networks.

4.1 picoCORBA

CORBA is now a mature distributed object architecture and a lot of effort has been devoted to embedded CORBA implementations (see section 2.2). Most of this previous work is influenced by MinimumCORBA [21], a reduced footprint specification from OMG. MinimumCORBA objects are completely standard compliant and they may also be built on full CORBA engines.

CORBA picoObjects (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.

Instead of a byte-by-byte message parser the FSM generated by the picoCORBA compiler will just compare the request message signature against the possible set of valid signatures. Some parts of the incoming message must be skipped from the signature calculation because they may be different on each invocation (arguments, request identifier, etc.) but all these details may be handled automatically by the picoCORBA compiler.

In order to simplify this procedure even further, we assume that the object identity string (`object_key` field) of every CORBA picoObject are exactly the same length. This assumption does 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.

We only implement messages for GIOP 1.0 even when clients may support later versions of the protocol. The mandated CORBA backwards compatibility ensures that this decission does not introduce interoperability problems.

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. The FSM generator may thus generate a message parser for both byte orders automatically (two valid signatures for each method implemented). The compiler may be instructed to assume a single byte order although this may introduce interoperability problems.

Any CORBA object should implement a set of common methods (the CORBA::Object interface). The compiler will just include the minimum set of messages needed not to compromise interoperability with standard tools. 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 CORBA picoObject even when no explicitly stated.

4.1.1 GIOP Messages

In this subsection we will discuss the design decisions of current picoCORBA implementation related to GIOP support. GIOP (General Inter-ORB Protocol [11]) is the only required protocol for all CORBA-compliant implementations.

GIOP 1.0 defines three types of client to server messages (Request, LocateRequest and CancelRequest) and three types of server-to-client messages (Reply, LocateReply and CloseConnection). In addition a MessageError message may signal error conditions.

Location messages are used to implement full location transparency. This feature is not needed on wireless sensors where physical location is almost as important as the functionality provided. Therefore we do not support that feature.

The GIOP specification explicitly allows servers to ignore CancelRequest messages. Therefore, picoCORBA ignores them without any interoperability concern. Likewise CloseConnection messages will never be generated because picoCORBA objects are considered "always on". Therefore picoCORBA objects may just implement version 1.0 Request and Reply messages without a significative loss of interoperability.

Request messages are also simplified as much as possible. For example, the `service_context` field and the `response_expected` fields are ignored. The CORBA picoObjects always generate replies even if they are not required. This is a standard compliant behaviour because the clients must ignore not requested replies. Besides the `object_key` field identifying the target object is assumed to be of fixed length. This is transparent to clients and it does not affect interoperability.

Reply messages are always generated when a correct request is received. Obviously, the reply message payload depends on the corresponding request message and also on the user procedure. The fields `service_context` and `request_id` are directly copied from the request message, and the `reply_status` field always contains the value `NO_EXCEPTION` since in picoCORBA neither exceptions nor location forward are supported.

An ErrorMessage should be sent by a receiver of an incorrect message. Since picoCORBA ignores any non-supported message, it also ignores malformed or incorrect messages and will never produce an ErrorMessage. For most cases this is not a severe limitation since CORBA assumes a reliable transport anyway.

4.2 PicoICE

ZeroC, Inc. developed a high quality distributed object framework called ICE (Internet Communication Engine) built upon the experience of CORBA. It implements a feature set unparalleled in any other free distributed object platform (object persistence, object migration, authentication, security, replication, deployment services, firewall gateways, etc.). Despite the current lack of support for deeply embedded platforms, ICE offers a few advantages over CORBA to reduce resource comsumption such as a simpler protocol.

Any ICE object must implement the `ICE::Object` interface by default. This interface provides minimal introspection capabilities (`ice_isA`, `ice_id`, `ice_ids`) and reachability test (`ice_ping`). All these methods are automatically generated by the picoICE compiler although we may optionally skip the introspection messages. Introspection is not used when objects are contacted using one-way transports (such as UDP).

4.2.1 IceP Messages

IceP is the protocol used for ICE object interaction. It defines two types of client to server messages (Request, BatchRequest) and two types of server-to-client messages (Reply, ValidateConnetion). In addition a CloseConnection message may be used by both peers to shut down a connection.

ValidateConnetion and CloseConnection messages are used only with connection oriented transport protocols. They may be ignored when sensors are using unreliable protocols (such as UDP).

PicoICE supports all IceP messages but one, *BatchRequest*. This does not introduce interoperability problems since batching is an optional optimization feature. It uses standard IceP headers but the `compressionStatus` field is always 0 (not allowed). Compression is also an optional feature which is not even applied to small messages (less than 100 bytes).

Request messages are similar to CORBA requests. There is a request identifier (`requestId`) used to match requests and replies which needs to be copied to the reply message. There is also an object identity similar to GIOP `object_key` field which is assumed to be constant length.

Reply messages contain results for two-way invocations, i.e. the output parameters and return value. Besides these messages carry a *replyStatus* field which picoICE sets always to 0 for correct invocations and to 7 (*unknown exception*) for unhandled requests. A more sophisticated error handling mechanism with more specific error reporting and exception raising may be implemented but it would require extra resources.

When an object (server-side) accepts a new incoming connection (when using a connection oriented protocol) it must first send a special message to acknowledge it. Likewise when a client stops sending requests to a server it sends a *CloseConnection* message. Server must then reply with another *CloseConnection* message. An ICE server may also send this message to tell the client that it will not accept new requests. ICE picoObjects reply to this message, however they do not emit them.

4.3 Basic interface for actors

In our discussion so far we described a completely general approach to the development of small distributed objects but we may simplify the implementation even further by specifying the set of allowed interfaces.

All actors (sensors or actuators) implement a very simple interface exposing an internal state. The state of a sensor is the value of the last measure of a physical magnitude. There are different interfaces depending on the data type they manage.

Depending on how actors interact with application objects we may identify four categories: **passive** actors must be queried synchronously using the `get` method, **active** actors are able to invoke a `set` message on another object (usually an event channel) passing the current state of the actor, **proactive** sensors are active sensors able to invoke `set` whenever a state change occurs, and **reactive** sensors are active sensors that will invoke `set` on-demand whenever a client invokes its standard `ice_ping` method. The semantics of `ice_ping` is therefore extended.

Passive actors require a two-way communication model while the active actors may also be used with a one-way communication model. Active actors implement an additional interface in order to comfigure remotely the destination of the `set` events.

4.4 Implementation results

Using the strategies and policies described so far we developed a set of prototypes for both picoCORBA and picoICE objects. Table 1 shows the size of several implementations of a picoObject for different platforms. It also shows the size of a server implemented on some other architectures.

Probably one of the most used microcontrollers in WSN platforms is the Atmel128. The size of a picoObject (ICE version) implemented on a Chipcon CC2420DB Zigbee node (platform based on an Atmel 128L microcontroller) is 1923 bytes. The Zigbee communications library (RF) accounts for 69% of the code.

5 Wireless sensor network services

Using standard middleware interfaces is a way to leverage the existing development tools and expertise, but the distinctive characteristics of WSN call for specific services. In this section we describe a set of services designed to ease the development of WSN applications.

Embedded middleware	Minimal server
TAO	1738KB+OS
nORB	567KB+OS
UIC/CORBA	35KB+OS
JacORB (Java)	243KB
ZEN (Java)	53KB+OS
MicroQoSCORBA (TINI)	21 KB
TinyLime (TinyOS mica2 platform)	16KB
TinyDB	58KB
picoCORBA (C)	5.2KB+OS
picoCORBA (Java)	5 KB+OS
picoCORBA (TINI)	4 KB
picoCORBA (PIC12C509)	415 words
picoICE (C)	5.4KB+OS
picoICE (PIC12C509)	503 words
picoICE (Atmel128)	1923 bytes
picoICE (Atmel128 + Zigbee library)	6184 bytes
picoICE (tinyOS mica2 platform)	11KB

Table 1: Size of a small server on embedded middlewares

5.1 Event channels

We use extensively the event generation and propagation service provided by the underlying middleware. For example, IceStorm, the ZeroC ICE *event channel* service, is able to use several transport protocols at same time in a transparent way. Each publisher or subscriber may select its own protocol individually.

However, it is not convenient to connect too many nodes to the same event channel due to scalability reasons. Therefore, several event channels (*topics* in ICE parlance) are used. Event channels impose minimal overhead and they may be federated by means of “links” to propagate events hierarchically. These links may be customized to filter-out events with a priority below a certain threshold.

Event channels and reactive sensors (see section 4.3) may be combined to generate powerful interaction models. For example, as a way to estimate the average temperature of a room we may use an event channel to propagate the standard `ice_ping` method to all the available temperature sensors. Then we may use another event channel to collect the `set` method invocations for all of them. The whole procedure is shown in figure 3.

5.2 Announcements

When a node is switched on (and also periodically), it invokes the `adv` method of a specific event channel named *ASD.announce*. The `adv` method publishes the remote object reference and an associated property service (see section 5.3). The whole invocation may be fully static and therefore it may reside in the device ROM. A node which is not even be able to invoke `adv` may delegate in an external agent.

Whenever a client or a service is interested in node advertisements it just needs to implement the iListener interface and subscribe to the *ASD.announce*

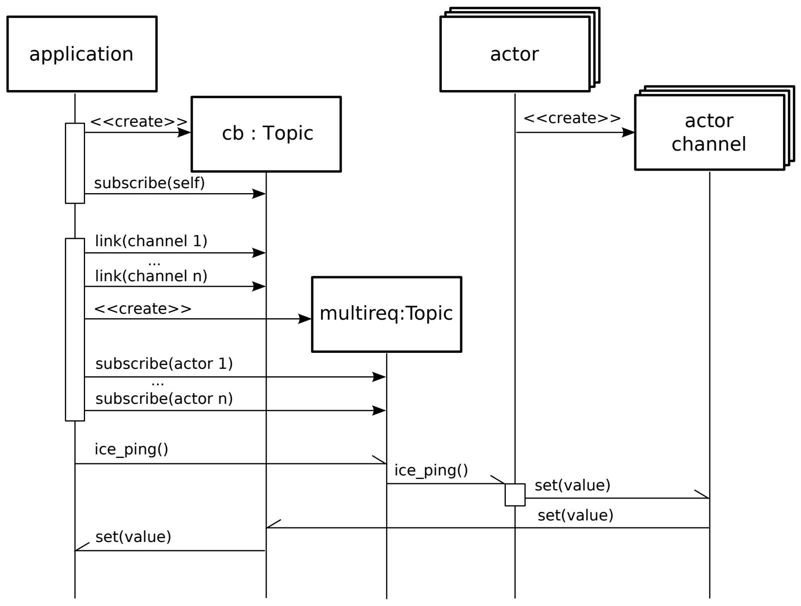

Figure 3: Sequence diagram for multi-requests.

channel. On each `adv` subscribers will get the object reference and then they may use the middleware standard introspection mechanisms to find out the interfaces implemented by the object. In spite of the high abstraction level of this announcement procedure it may be implemented on very simple devices with an identical behaviour respect to a conventional object.

5.3 Properties

As seen above, the `prop` argument in the `adv` method is an object proxy (the *property server*). This is a conventional object which keeps a list of property values for the announced object. Properties are specified by means of a textual key and a values that may range from a scalar types such as an integer or a string to binary drivers or Java applets. They are always optional information, useful for initial deployment and system configuration applications, but system functionality is never affected by the lack of a property server.

There are several deployment alternatives available: (1) the property server may refer to an external dedicated property service which manages properties for many nodes; (2) or it may be implemented by the node itself; (3) or there may even be no property server for a given node.

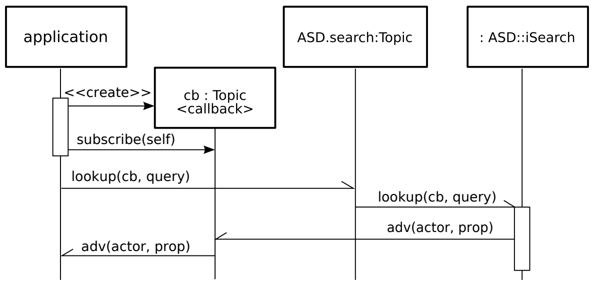

Figure 4: Sequence diagram for service lookup.

5.4 Service Lookup

When an application needs to find an object providing a particular service it needs a callback object (it may even be an event channel to share the results). Then the application invokes the `lookup` method on the *ASD.search* event channel passing the requested property values and the callback proxy.

The actors (subscribed to the *ASD.search* channel) matching the criteria will invoke `adv` method of the callback proxy. A sequence diagram of this procedure is shown in figure 4. Note that the figure shows a callback object implemented as an event channel. Therefore other applications interested in the results may subscribe to it.

5.5 Network deployment

Deployment of WSN has been traditionally approached by means of an application level gateway. Multi-hop routing algorithms coexist with gateway nodes which store data collected from the WSN and provides such information to the remaining infrastructure. They behave as a bridge between different technologies used in WSN (i.e. Zigbee) with technologies used in the remaining of the system (i.e. Ethernet or 802.11 based networks).

Meanwhile most previously proposed architectures assume some application level components runing in the gateways just for integration purposes (i.e sensor abstraction layer in Sensation project or the base station host in TinyLIME). Our approach reduces the role of the gateway to a simple bridge between different technologies which usually implements just header rewriting.

6 Conclusions

Standard distributed object platforms offer an attractive development model for WSN applications. Traditional implementations focus on flexibility and features but there is a broad range of design tradeoffs to be explored on specific environments. The picoObjects and the hardware middleware proposed in this article meet the needs of powerful wireless sensor nodes without compromising the interoperability with standard services.

Besides WSN applications deal with extremely dynamic environments that need specific support services. We described a set of services ranging from event propagation to discovery services that cooperate to build a robust infrastructure for highly scalable WSN applications. Built on top of a traditional RMI concept, these services may be defined at a very high level of abstraction.

Our prototypes show that the proposed implementation strategies lead to very low footprint, two orders of magnitude lower than some previous approaches to middleware embedding and also lower than other WSN middlewares. Our approach vastly reduces the complexity of network gateways. Besides the hardware object implementation extends the range of WSN applications to more demanding needs by means of a low-cost FPGA device.

The proposed design flow achieves better orthogonalization of concerns by separating the WSN application development, the embedded code development, and the application deployment. WSN application developers may use standard distributed development tools and design flows. Embedded code developers use a very similar design flow starting from the same interface specification. Deployment is orthogonal to the application development and may be changed interactively using a flexible distributed property service.

Future developments will focus on higher level services for improved scalability, object migration, fault tolerance, real-time characterization, and quality of service guarantees.

References

[1] Mauri k., Marko H. and Timo D. A Survey of Application Distribution in Wireless Sensor Networks, *EURASIP Journal on Wireless Communications and Networking*, 2005, Vol 5, pp. 774-788.

[2] Tilemahos H., George A., Vassileios T., Odysseas S. and Stathes H. Sensation: A Middleware Integration Platform for Pervasive Applications in Wireless Sensor Networks , *Proc. of the 2nd European Workshop on Wireless Sensor Networks*, 2005, pp. 366-377.

[3] Curino, C. and Giani, M. and Giorgetta, M. and Giusti, A. and Murphy, A. L. and Picco, G. P. TinyLIME: bridging mobile and sensor networks through middleware, *Third IEEE International Conference on Pervasive Computing and Communications* 2005; pp. 61-72.

[4] C. Intanagonwiwat, R. Govindan, D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks, *in Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking* 2000; pp. 56-67.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. System Architecture Directions for Networked Sensors. *Proceedings of Ninth Interna-*

tional Conference on Archi- tectural Support for Programming Languages and Operat- ing Systems (ASPLOS), 2000; pp 93-104.

[6] Sam M., Michael J.F., Joseph M.H, Wei H. TinyDB: An Acqusitional Query Processing System for Sensor Networks, *ACM Transactions on Database Systems* 2005; 30(1), pp.122-173.

[7] 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) and Networking (ICN), World Scientific Publishing* 2002; pp 165-176.

[8] J.H. Park, M.J. Lee, S.j. Kang. CORBA-based distributed and replicated resource repository architecture for hierarchically configurable home network, *Journal of Systems Architecture* 2004; Vol 51 pp. 125-142.

[9] J. Oh, J. Park, G. Jung, S. Kang. CORBA based Core Middleware Architecture Supporting Seamless Interoperability between Standard Home Network Middlewares, *IEEE Transactions on Consumer Electronics* 2003; Vol 49, No 3, pp 581-586.

[10] M. Henning, M. Spruiell. Distributed Programming with Ice, 2006, available online at http://www.zeroc.com/ (Last visited 26-2-2007).

[11] Object Management Group, The Common Object Request Broker: Architecture and Specification, ed. 2.3, June 1999. Available in http://www.omg.org/ (Last visited 26-2-2007), document id: 98-12-01.

[12] Sun Microsystems, Jini Architecture Specification, ed. 1.2, available online at http://www.sun.com/ (Last visited 26-2-2007).

[13] R. Gupta, D.P. Agrawal. Jini Home Networking: A Step toward Pervasive Computing, *Computer* 2002; pp. 34-40.

[14] D. Puccinelli, M. Haenggi, Wireless Sensor Networks: Applications and Challenges of Ubiquitous Sensing, *IEEE circuits and systems magazine* 2005; Vol 5, No 3, pp 19-31.

[15] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the Physical World, *IEEE Personal Communications* 2000; vol. 7, no. 5, pp 10-15.

[16] A. Boulis SensorWare Users & Programmers Guide Available at http://sensorware.sourceforge.net/ (Last visited 26-2-2007). March 2004.

[17] A. Boulis, C.-C. Han, M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks, *In Proceedings of MobiSys* 2003; pp. 187-200.

[18] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems, *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* 2003; pp. 107-118.

[19] K. Romer , Programming Paradigms and Middleware for Sensor Networks, *GI/ITG Workshop on Sensor Networks Avalaible* 2004; pp 49-54

[20] S. Li, S.H. Son, and J. A. Stankovic. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks, *Information Processing in Sensor Networks: Second International Workshop* 2003; LNCS Vol. 2634, pp. 502-517.

[21] Object Management Group, *Minimum CORBA Specification*, ed. 2.3, August 2002, available online at (Last visited 26-2-2007) http://www.omg.org/, document id: 02-08-01

[22] M. Roman, A. Singhai, Integrating PDAs into Distributed Systems: 2K and PalmORB, *In Proceedings of International Symposium on Handheld and Ubiquitous Computing* 1999; pp. 137-149.

[23] M. Román, Fabio Kon, Roy H. Campbell, Reflective Middleware: From Your Desk to Your Hand, *IEEE Distributed Systems Online* 2001; 2(5).

[24] Fabio Kon, F. Costa, G. Blair, Roy Campbell. The Case for Reflective Middleware, *Communications of the ACM: special issue in adaptive middleware* 2002; Volume 45, Issue 6, pp 33-38.

[25] M. Roman, M. Dennis, Mickunas, Fabio Kon and Roy Campell. LegORB and Ubiquitous CORBA, *In Proceedings of IFIP/ACM Middleware'2000 Workshop on Reflective Middleware* 2000; pp. 1-2.

[26] PrismTech, OpenFusion e*ORB Real-time Embedded Whitepaper, available online at (Last visited 26-2-2007) http://www.prismtechnologies.com/.

[27] V. Subramonian, G. Xiang. Middleware Specification for Memory-Constrained Networked Embedded Systems, *In 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* 2004; pp 306-313.

[28] W. Nagel, N. Anderson. A Protocol for Representing Individual Hardware Devices as Objects in a CORBA Network, *Real-time and Embedded Distributed Object Computing Workshop* 2002.

[29] Haugan, Olav. Configuration and Code Generation Tools for Middleware Targeting Small Embedded Devices, M.S. Thesis, Dec 2001, Washington State University.

[30] Rodrigues, G., Ferraz, C. A CORBA-Based Surrogate Model on IP Networks, *In Proceedings of the 21st Brazilian Symposium on Computer Networks* 2003.

[31] D.C. Schmidt and C. Cleeland. Applying Patterns to Develop Extensible ORB Middleware, *IEEE Communications Magazine* 1999; vol. 37, no. 47, pp. 54-63.

[32] A. Dunkels, Full TCP/IP for 8-Bit Architectures, *In Proceedings of the first international conference on mobile applications, systems and services* 2003; pp 85-98.

[33] A. Dunkels, Juan Alonso, and Thiemo Voigt. Making TCP/IP Viable for Wireless Sensor Networks, *In Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN)* 2004.