Web Services for deeply embedded extra low-cost devices*

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, Felix.Villanueva, Francisco.Moya, Fernando.Rincon, Jesus.Barba, JuanCarlos.Lopez}@uclm.es

Abstract. This paper describes a new approach to implement Web Services in embedded devices connected to Wireless Sensor Networks. The sensor/actuator node is able to process standard requests (XML-RPC and SOAP), perform an action and generate a valid response.
These stand-alone nodes show good interoperability with standard Web Services using just a transport protocol gateway.

1 Introduction

While interoperability may be achieved by means of low cost TCP/IP implementations, interoperability at the application layer continues to be a key problem [4].

Web Services emerge as an interoperable, language and platform independent solution to access sensor services thought Internet. Being able to introduce Web Services directly in the WSN devices would allow deploying application independent gateways. This implies a set of interesting advantages as we will show below.

Obviously, the use of SOAP [5] in wireless sensor networks introduces considerable overhead compared to most binary protocols. Therefore it may not always be appropriate in some application domains where power consumption or latency are critical. Our prototype implementation was focused on emergency light control, which shows a number of characteristics compatible with this approach:

- Nodes do not have power supply problems because they are attached to the power line.
- Nodes are static. There are no mobile nodes in the network.
- Interaction across the WSN is quite limited. In fact, the nodes will only provide autochecking notifications on request so that delay and bandwidth are not major bottlenecks.

* This work was supported by ERDF, the Regional Government of Castilla-La Mancha, and the Spanish Ministry of Science and Innovation under grants PAI08-0234-8083 (RGrid), TEC2008-06553 (DAMA), and Hesperia CENIT.

– The configuration procedure should be as simple as possible in order to reduce installation cost. We initially aim at per-node remote configuration. Global configuration would be handled as an external system-wide service.

In this case, or any other field with similar properties, Web Services seem to be a good approach to solve the problem of interoperability at the application layer.

2 Related work

Most of the former approaches take the raw data from wireless sensor networks using proprietary protocols and export them through Web Services residing in a gateway [1, 3]. The web service is actually running in the gateway and not in the WSN device. Each new application would require specific developments, and each web service becomes a wrapper or a facade for the binary protocols used in the WSN.

Our approach, as we will see later, uses generic gateways (independent of the wireless sensor network devices deployed) and reduces configuration procedures required to describe and publish the WSN nodes to a specific network. We aim at removing the need of an intermediate application-level proxy such as the Sensor Collection Service described in [1].

In [2] a framework has been developed to facilitate the use of SOAP in WSN focusing on reducing the overhead (e.g reducing the number of messages) on the network by means of data aggregation techniques. Their implementation is done in the NS2 network simulator and we are not aware of any implementation for actual devices.

Other approaches [8] try to embed the Web Services architecture in low-cost devices reducing the size of the protocol stack (TCP/IP, XML and SOAP processor, etc.). These low-cost devices exceed the average capacity of many wireless sensor networks devices.

In this paper we propose a different way to embed a minimun web service in a WSN device which follows the general principles described in [11]. Instead of reducing existing implementations of the Web Services protocol stack, we are going to define the smallest feature set that a web service needs to provide and build it up from there.

3 Embedding web services, a bottom-up approach

Although it is important that each device looks like a web service, it is not essential that they are real Web Services. If devices are able to generate coherent replies when they receive predefined request messages then the system will work as expected. For a given WSDL [7] specification these request and reply messages are completely specified by the communication protocol (SOAP or XML-RPC).

Let us analyze an HTTP embedded SOAP request and the corresponding reply for a very simple interaction (e.g. get the status of an emergency light).

We will show the full SOAP messages to make it easier to follow the behavior of the automaton of figure 1.

```
POST / HTTP/1.0
Host: localhost:8080
User-agent: SOAPpy 0.12.0 (pywebsvcs.sf.net)
Content-type: text/xml; charset="UTF-8"
Content-length: 336
SOAPAction: "get"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<get SOAP-ENC:root="1">
</get>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

It seems clear that SOAP is a verbose protocol where most of the information is in the `Body` part of the message. This message can be obtained from the WSDL specification:

```
[...]
 <message name="getRequest">
  </message>

  <message name="getResponse">
    <part name="retval" type="xs:boolean" />
  </message>
[...]
<operation name="get">
      <input message="getRequest" />
      <output message="getResponse" />
</operation>
[...]
```

We implement an ad-hoc parser from the WSDL specification which recognizes the request and builds, in a dynamic way, an appropriate answer to such a request:

```
HTTP/1.0 200 OK
Server: <a href="http://pywebsvcs.sf.net">SOAPpy 0.12.0</a> (Python 2.5.2)
Date: Fri, 21 Nov 2008 10:52:17 GMT
Content-type: text/xml; charset="UTF-8"
Content-length: 501

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<getResponse SOAP-ENC:root="1">
<Result xsi:type="xsd:boolean">False</Result>
</getResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Typical sensors usually expose a very simple interaction model. Therefore we propose a set of very simple WSDL interfaces to access the data provided by the nodes. We define a get/set interface that may be used by the clients to obtain the current value of a sensor (`get`) or manipulate and actuator (`set`). Of course, the approach is not limited to this interface, but it illustrates perfectly the proposed solution.

We focus on minimizing the amount of memory required to parse the messages in order to be able use WSAN devices with limited resources. To achieve this, we use several techniques:

- We implement an ad-hoc finite state machine in order to recognize a request and build up the answer. The parser removes redundant whitespace, and XML comments in order to feed a canonical XML stream into the automaton.
- Instead of saving and comparing all possible tags that a request message can have, we calculate a CRC using the CCITT-CRC 16 bit algorithm [10]. In the parser, we only compare CRC values of the XML tags with the precalculated CRC for each request available in the WSDL specification.
- We do not store whole messages. Each message is parsed incrementally as it is being received, much like a SAX parser, but using an ad-hoc parser to reduce overall resource consumption.

For example, for the request message shown above, we implement the finite state machine shown in figure 1. Basically, we calculate the CRC for each XML tag (delimited by '`<>`'). From the `Init` state, we start to parse the input data ignoring the HTML header. This header does not include relevant information from the point of view of the WSAN devices.

The parser ignores all incoming data until the tag `<?xml version="1.0" encoding="UTF-8"?>` is recognized (the calculated CRC matches the expected CRC for this tag). This event triggers the transition to the `Start` state, which will discriminate among the available operations. Input data will include some encoding tags inmediately followed by the operation being invoked, which helps us to recognize whether the request is a valid verb (*set* or *get* in our example). Depending on the CRC calculated at the expected positions of the incoming data stream there will be a transition to the state where the corresponding arguments are read. Operations without arguments will lead to a transition to the `Tail` state which validates the remaining of the request message.

Operations with parameters will go through intermediate parameter parsing states. In our example, in the `Req Set` state, the automaton will parse the parameter attached to the *set* operation and then it will trigger a transition to the `Tail` state. Finally, after a completely validated request is received we build up the response message specifically for each request. Note that there is no need to store which method is being invoked since the tail of each request is different and may be used to discriminate among the response generators.

In each of the response generation states (`Resp Set` and `Resp Get` in our example), the user needs to provide an appropriate routine which executes the validated request and builds the appropriate response.

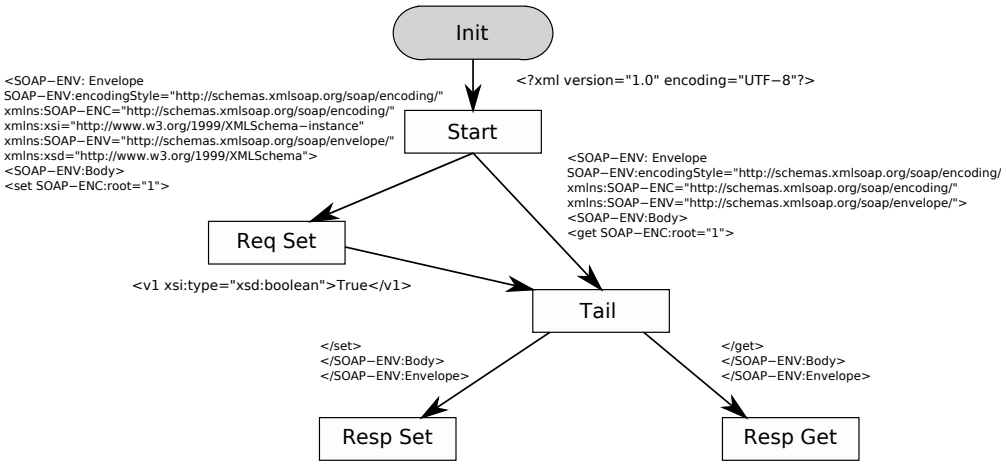

Fig. 1. Finite state machine for `get()` and `set()` SOAP messages

Although not explicitly stated in figure 1 (for the sake of clarity) whenever the incoming tags do not match any of the expected CRCs the state machine is reset to the initial state `Init` and the connection is closed. This provides a reasonable protection against faulty clients.

3.1 Compiler

The design and implementation of the finite state machine is a tedious and error prone task. Therefore we are building a compiler that takes a WSDL specification and a service definition and automatically generates the appropriate finite state machine for the corresponding messages from the interfaces defined in the WSDL file.

The compiler (see figure 2) takes four files as inputs:

- The WSDL interface declaration. The services provided by the WSN node must adhere to the interfaces specified in this file.
- The servant (method implementation). This file contains the application specific code. Usually this includes code to access to the underlying hardware and read or write a physical transducer.
- Service definition. The programmer defines in this file which concrete services the node holds and the interface (`portType`) exposed by each sensor or actuator in the node. We use a very simple syntax defining services and basic interaction events. The following listing shows a little example.

```
uses "DUORW.wsdl";

local myLocalEP("xbow -h 0x0001") {
     DUOIBoolRW_Service svc1;
}
```

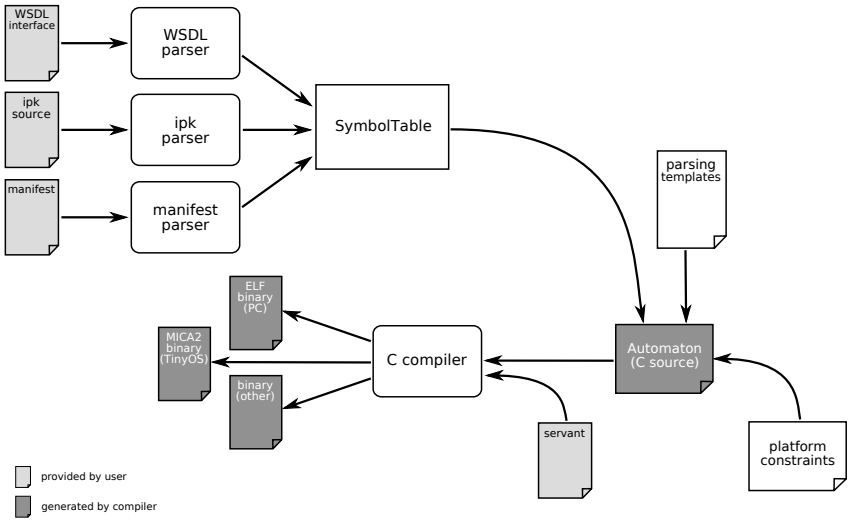

Fig. 2. Compiler block diagram.

```
remote myRemoteEP("xbow -h 0x0002") {
     DUOIBoolRW_Service svc2;
}

repeat(5) {
     svc2.set(svc1.status(), svc1.endpoint);
}
```

This example defines both a local and a remote web service endpoint. It also states that a periodic request must be done in the remote service sending the status and the endpoint of the local service.

– A manifest file which contains the interfaces for user-provided procedures, such as the procedures to read hardware sensors. The *status* procedure of the listing above is one such example:

```
private DUOIBoolRW status(11) {
    output = bool;
};
```

The compiler is able to generate generic parsers that may be compiled on several platforms. This include support for the MICA2 (Crossbow) on the TinyOS [13] operating system, and also for OS-less implementations for general purpose microcontrolers such as ATMega128, 8051 or even mid-range Microchip devices.

4 Network Architecture

In our application field (emergency lighting), a control center usually manages and monitors several buildings distributed in a large geographical area. Buildings with hundreds or even thousands of emergency lights are quite common

(airports, museums, etc.). Each installation is connected to an external network (e.g. Internet) by means of one or more gateways (figure 3).

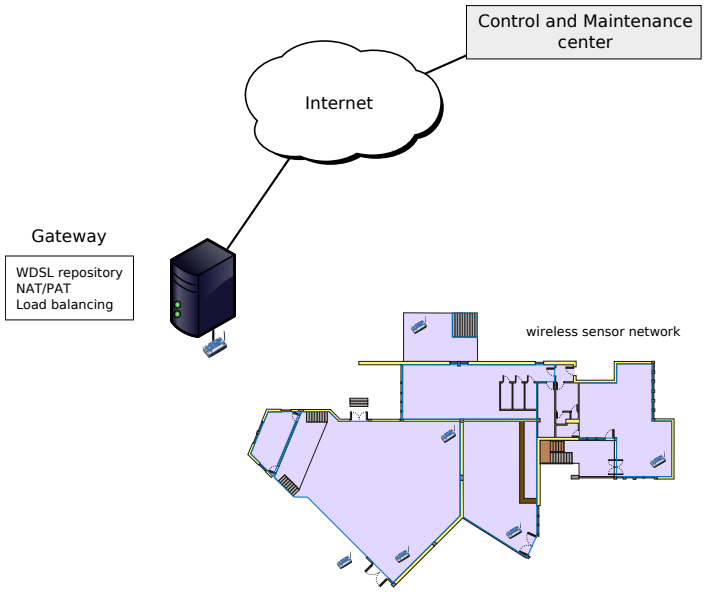

Fig. 3. Physical network topology

Each gateway performs three main tasks:

- It encapsulates SOAP messages from Internet transport protocol (typically TCP protocol) in the WSN transport protocol. The reverse process is also needed for outgoing messages (encapsulation of SOAP messages from WSN transport protocol to Internet). This task is shown in figure 4.
- It must implement a bidirectional correspondence between TCP ports and node IDs. This is a crucial aspect for avoiding hand-made configuration procedures. When we install a WSN node, the gateway must assign a TCP port to its node ID.

The association established between the TCP port and the node ID must be represented in the WSDL. A WSDL file represents the interface for users of the service and also some properties of that service, for example, its location (endpoint).

We embed both the WSN location and the TCP port assigned in the WSDL tag `soap:address`, as shown below. The gateway needs to know the WSN node ID of each WSN device available in the network and it generates the corresponding WSDL file. This task may be accomplished without human intervention using a modification of ASDF [12] (our service discovery protocol for WSN).

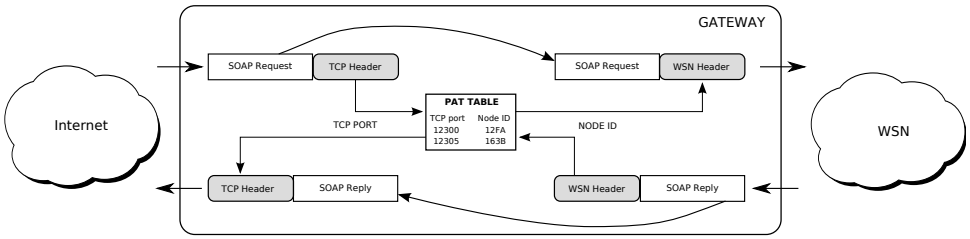

Fig. 4. Simplified gateway process

When a WSN node is attached to the network it sends an advertisement with the name of the interfaces that it implements and its node ID. The gateway builds a WSDL file from a template which contains the service interface and the service implementation. In the service implementation section we may include some common information about the environment such as human readable location, node ID, etc.

Here is an example:

```
[...]
<!-- Service to export -->
  <service name="ExitDoor_Service">
    <port name="IBoolRW_Port" binding="IBoolRW_Binding">
      <soap:address location="http://example.com:7890"/>
    </port>
  </service>
[...]
```

With this WSDL file, the services may also be located by means of UDDI protocol. The process of publishing any Web Service associated to a WSN device in a UDDI registry is similar to any other Web Service, namely, the web service interface like UDDI t-model structure and the Web Service implementation like a UDDI business Service.

Additionally, it is possible to install more advanced services in the gateway to provide additional features such as logging, authorization, etc.

The gateway may be able to generate the associated WSDL file from a set of templates for each of the interfaces implemented by nodes in the network. There are two possible solutions:

- Templates for each interfaces are stored in the gateways. Whenever new nodes are installed in the WSN their interfaces must be added to the template repository of the gateways (if they where not already there). This situation introduces coupling between gateways and applications deployed in the SAN, but do not affect nodes themselves.
- When a node is advertised through the network, the gateway may request the template of the interfaces implemented in the node. This is easily achieved with a small modification of ASDF storing the WSDL template in the node

and implementing an operation to retrieve the template. This is more expensive for the nodes in terms of the amount of required FLASH/ROM, but it allows simpler gateways, fully independent of the application.

5 Prototypes

Using the strategies and policies described so far we developed a set of prototypes on widespread WSN hardware.

Probably one of the most used microcontrollers in WSN platforms is the Atmel ATMega128 with the TinyOS operating system. OS-less devices are also quite common.

We compare here the size of different Web Services implementations. In a first step, we implement a basic get/set service (see section 3), using two lightweight general-purpose libraries: `csoap` [14] and `libxml-rpc` [15]. Then we implement the same service using our approach on three platforms: a PC, a MICA2 running TinyOS and on an OS-less AVR microcontroller. The size of the resulting binaries are shown in the table 1. Notice that:

- All the x86 prototypes are statically linked and they run on a conventional PC with Debian GNU/Linux. All of them require an operating system whose size is not taken into account.
- The x86 Embedded WS prototypes are pure C programs with standard socket support.
- The TinyOS and AVR prototypes include all the required components. The data refers to the binary file we install in the device. The AVR prototype does not use an operating system but just a simple cyclic executive runtime, much smaller than TinyOS although it lacks most of TinyOS features.

Software	Platform	Middleware	Binary size	Other
C-SOAP	x86	SOAP	1,731,508	OS
libXML-RPC		XML-RPC	768,536	OS
Embedded WS	x86	SOAP	507,676	OS
		XML-RPC	504,772	OS
	TinyOS	SOAP	35,150	
		XML-RPC	11,216	
	AVR	SOAP	1,068	ZigBee (4.261)
		XML-RPC	1,182	ZigBee (4.261)

Table 1. Size of the implementation (in bytes) of a simple emergency light controller as described in section 3.

The source code for all these tests is available in our EWS webpage [16]. The current reference implementations and other useful data may be found there.

6 Conclusions

In this paper we presented a very low-footprint implementation of web services for wireless sensor actuator networks. As far as the authors know this is the most reduced implementation of this type of architecture in WSAN.

Introducing web services directly in the WSAN devices allows building generic and application independent gateways. In this way, we enable the deployment on WSAN devices with minimun configuration procedures.

Some prototypes for SOAP and XML-RPC have been implemented showing the feasibility of our approach. Our current efforts are focused on improving and testing the compiler to allow the generation of complete ready-to-use sensor nodes.

References

1. T. Kobialka, R. Buyya and C. Leckie. *Open Sensor Web Architecture: Stateful Web Services* In Proceedings of the Third International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP) 2007
2. A. Al-Yasiri and A. Sunley. *Data aggregation and middleware in wireless sensor networks* In Sensor and theory applicatios XIV (SENSOR07), Journal of Physics: Conference Series 76, 2007.
3. Arch Rock Corporation. *Arch Rock Primer Pack product gateway datasheet*. Product Catalog, 2007.
4. B. Priyantha, A. Kansal, M. Goraczko and F. Zhao. *Tiny Web Services for Sensor Device Interoperability* At International Conference on Information Processing in Sensor Networks (IPSN), 2008.
5. M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau and H. F. Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework*.W3C Recommendation, Available at www.w3.org, 2003.
6. D. Winer. *XML-RPC Specification*. UserLand Software, Inc, Available at http://www.xmlrpc.com/, 2003.
7. D. Winer. *WSDL Specification*. World Wide Web Consortium. Available at http://www.w3.org/TR/wsdl, 2001.
8. J. Helander and Y. Xiong. *Secure Web services for low-cost devices*, Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005.
9. J. Helander. *Deeply Embedded XML Communication, Towards an Interoperable and Seamless World*, EMSOFT, 2005.
10. P. Koopman and T. Chakravarty, *Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks* Dependable Systems and Networks, 2004.
11. D. Villa, F.J. Villanueva, F. Moya, F. Rincón, J. Barba, J.C. López. *Embedding a general purpose middleware for seamless interoperability of networked hardware and software components* Grid and Pervasive Computing, GPC 2006, Taiwan May 2006. Lecture Notes in Computer Science 3947.
12. D. Villa, F.J. Villanueva, F. Moya, F. Rincón and J. Barba J.C. López *Minimalist Object Oriented Service Discovery Protocol for Wireless Sensor Networks*,International Conference on Grid and Pervasive Computing, 2007.

13. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. *System Architecture Directions for Networked Sensors*. Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.
14. csoap client/server SOAP library in pure C, available online at http://csoap.sourceforge.net/.
15. XML-RPC for C and C++, A lightweight RPC library based on XML and HTTP, available online at http://xmlrpc-c.sourceforge.net/.
16. EWS Webpage. Available online at http://arco.esi.uclm.es/en/ews.