QUESTRA

Web Services in Embedded Systems

July 2001

WHITE PAPER

Contents

Introduction

There is a long history of technologies that were originally designed for scientific and enterprise computing finding their way into embedded systems. In fact, you can go back and look at the first electronic computer, the ENIAC. ENIAC was designed to calculate artillery and missile trajectories , while today's embedded systems in the missiles not only calculate but also control those same trajectories. Other examples of technologies originally developed for mainframe and server systems that have filtered down to embedded systems include RS-232, virtual memory, instruction/data caches, operating systems, graphical displays and of course everyone's favorite technology, the Internet.

Perhaps the only change in this trend over the years is that the speed of this technology adoption has accelerated. That is why any organization that wants to stay abreast of developments in the embedded systems world should keep at least one eye on what is happening in the enterprise and business computing space. Perhaps the biggest developing story in the enterprise space is Web Services. It is unusual to see the likes of Microsoft, Sun, IBM, Oracle, HP and others agree on anything, but they all agree that Web Services will be the native language of business applications. When you hear about Microsoft.NET, Sun ONE, HP's e-services and IBM's WebShpere, you are hearing about Web Services. Most of these organizations speak of Web Services in the context of Business-to-Business (B2B) and Business-to-Consumer (B2C) information exchange and e-commerce. As you will discover in the following article, Web Services are just as powerful for connecting e-appliances and other distributed intelligent assets into the business enterprise to provide such valuable services as automatically generating service requests, performing remote diagnostics and automatic consumables reordering.

Web Services

A Web Service is a programmable component that provides a business service and is accessible over the Internet. Web Services can be standalone or linked together to provide enhanced system functionality. Buying airline tickets, accessing an online calendar and obtaining tracking information for your overnight shipment are all business functions that have been exposed to the outside world as Web Services. Web Services consist of a set of methods that operate on messages containing either document-oriented or procedure oriented information. An architecture that is based on Web Services is the logical evolution from a system of distributed object-oriented components to a network of services, providing a loosely couple infrastructure that enable crossenterprise integration. Web services differ from existing component object models and their associated object model specific protocols , such as CORBA & Internet Inter-ORB Protocol, Component Object Model (COM) & DCOM, and Java & Remote Method Invocation (RMI) in that the distributed components are interfaced via non-object specific protocols. Web services can be written in any language and can be accessed using the familiar, and firewall friendly Hypertext Transport Protocol (HTTP).

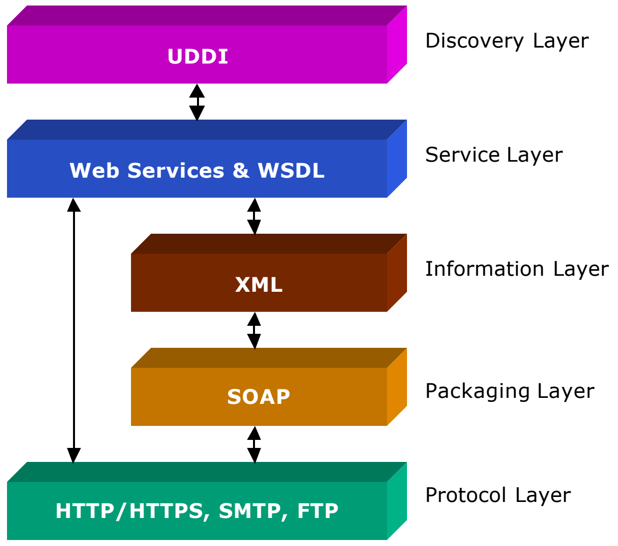

Figure 1 Web Services Stack

As shown in Figure 1, Web Services consist of multiple layers that, when stacked together, form the basis for a standard mechanism for discovering, describing and invoking the functionality provided by a stand alone Web Service.

Protocol Layer - HTTP

Starting at the bottom of the layered architecture model in Figure 1, any of the standard Internet protocols may be used to invoke Web Services over the network. The initial definition focuses specifically on HTTP/1.1 (and HTTPS (HyperText Transport Protocol Secure) protocols. HTTP/1.1 is a text-based, "request-response" type protocol that specifies that a client opens a connection to a server, then sends a request using a very specific format. The server then responds and, if necessary, keeps the connection open. Other request/response style transports, such as File Transfer Protocol (FTP) and Simple Mail Transport Protocol (SMTP) can also be used but are not yet defined in the standards around Web Services.

Packaging Layer - SOAP

Simple Object Access Protocol (SOAP) is a lightweight protocol designed for the exchange of information. Focused on distributed, decentralized environments, it provides a framework to invoke services across the Internet and provides the mechanism for enabling cross-platform integration independent of any programming language and distributed object infrastructure. SOAP represents the evolution of *xml-rpc* and has been adopted as an Internet standard to W3C. SOAP is text (XML) based and can run over HTTP, making it better able to operate in the face of firewalls than DCOM, RMI or IIOP. SOAP is also simpler to implement on an embedded device than developing an ORB.

SOAP does not define a programming model or implementation; instead it defines a modular packaging model and the encoding mechanisms for encoding data within modules. This allows SOAP to be used in any number of systems ranging from message passing systems to remote procedure calls. As shown in Figure 2 a SOAP message is XML encoded and consists of three parts:

- The SOAP envelope that provides the framework for packaging message information.
- The SOAP encoding rules that defines how it should be processed.
- The SOAP RPC representation that defines how to represent remote procedure calls and responses.

The SOAP specification also defines bindings to transport SOAP messages using the HTTP protocol. SOAP messages are uni-directional. Individual messages are typically combined to form a request/response mechanism. Figure 2 illustrates an example SOAP message. This example shows the request message for an "Event" Web Service – where the information of the message is XML content, and the packaging is accomplished using SOAP. In the example in Figure 2, the result to the SubmitEventRequest is returned as another SOAP message, shown in Figure 4. This message is packaged as part of the HTTP response.

```
POST /a2b/EventService HTTP/1.1
Host: a2b.example.questra.com
Content-Type: text/xml; charset="utf-8‟
Content-Length: nnnn
SOAPAction: /a2b/EventService#SubmitEvent

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header>A2B Header</SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <SubmitEventRequest>
       <Source>Copier54321</Source>
       <Description>MotorFailure</Description>
    </SubmitEventRequest>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2   SOAP Request Message Example

The SOAP message exchange models requires that applications receiving a SOAP message execute the following sequence of actions:

1. Identify all the components of the SOAP message that are intended for this particular application. Applications may act as SOAP intermediaries and pass parts of the message on to other applications.
2. Verify that all the mandatory parts specified in the SOAP message are supported by the application and process them accordingly.
3. If the SOAP application is not the end destination of the message it should remove all the parts that the application consumes and then forward the message to the next application the message is intended for.

The SOAP encoding style uses both scalar types (strings, integers, etc.) and compound types such as structures and arrays. These types appear as elements in an XML document. The data types defined in the XML Schema specification along with types derived from those data types can be used directly as SOAP elements. Figure 3 shows an example of a compound data type.

```
<xsd:complexType name="Event">
        <xsd:element name="Source" type="xsd:string"/>
```

```
                  <xsd:element name="Description" type="xsd:string"/>
                  <xsd:element name="MemberId" type="xsd:string"/>
                  <xsd:element name="GeneratedAt" type="xsd:timeInstant" minOccurs="0"/>
                  <xsd:element ref="EventInfoList" minOccurs="0"
            </xsd:complexType>
```

Figure 3 Compound SOAP Data Type

Any robust messaging system must expect that faults can occur and SOAP is no exception. SOAP has a built in Fault element that is used to carry error and status information. SOAP defines several built-in fault codes to handle such errors as version mismatches, not understanding some elements marked as required and incorrectly formed messages. Being based on XML makes the fault reported system of SOAP highly extensible and quite flexible.

The SOAP specification details the mechanisms for using SOAP for making remote procedure calls (RPC). The example shown in Figure 2 actually invokes a remote procedure (in this case a Web Service called EventService) by packaging the parameters required by the procedure as a structure. The response to a method invocation is also modeled as a structure containing the return value and possibly the parameters in the same order they were passed in.

```
HTTP/1.0 200 OK
Content-Type: text/xml
Content-Length: 316
<?xml version="1.0" encoding="UTF-8"?>

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Header>A2B Header</SOAP-ENV:Header>

   <SOAP-ENV:Body>
      <SubmitEventResponse>
         <Status>OK</Status>
      </SubmitEventResponse>
   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Figure 4 SOAP Response Message Example

It is important to note that Web Services and SOAP are two different things. SOAP is simply one way to package and bind the information required to invoke a Web Service. It happens to be the method that is most commonly associated with Web Services, but Web Services can be invoked using other encoding techniques, such as simple URL-encoded messages for bandwidth constrained scenarios. In addition, SOAP itself is not strictly relegated to Web Services. SOAP can be used as an access mechanism for any type of remote objects or procedures or a just a simple message passing mechanism.

Information Layer – XML

XML (eXtensible Markup Language) is a meta-language that enables cross-platform data interchange using a standard method for encoding and formatting information. Unlike HTML (HyperText Markup Language), XML affords you the freedom to publish useful information not only about how your data is structured, but also what your data means (i.e., its context). In addition, significant benefits are accrued through maintaining an XML document's structure, content and presentation as three distinct components. For example, part of the content of an HTML document may look as follows –

<b> Motor Failure</b>

this instructs a browser to display the text string "Motor Fault" as bold text. HTML is purely about formatting and display. XML takes it much further and provides information about *what* the content is describing, not just about how it looks. For example, an XML document may take the same text and but actually apply a datatype to is as follows –

</FaultType> Motor Failure </FaultType>

This gives the interpreter of this document a much dearer understanding of what the text actually is trying to signify. For more information than anyone could possibly absorb about XML direct your browser to http://www.xml.org.

It should be noted that there are those out there that would tell you that all you need to do is support XML in your devices and you have solved all of the world's problems. XML is not a panacea, it is a markup language. Supporting XML alone does not magically integrate you into a host of business applications or make for a complete solution. XML may provide a description of an event as a fault, but it does not provide the device or business logic that says what to do when that fault occurs or how to ensure that the fault indication is properly captured in one or more enterprise applications. That is why the higher level concept of Web Services, which are based on XML is so important – Web Services provide access to this logic.

Services Layer – Web Services and WSDL

The interface to Web Services are defined in the XML based Web Services Description Language (WSDL) which provides all of the information necessary for an application to access the specified Web Service.

A WSDL document is an XML based description of a Web Service. WSDL takes great pains to promote reusability. Several abstract and concrete elements are combined to define the functionality and access mechanisms of a Web Service. The following list enumerates the elements used when defining a Web Service.

- Types – containers for data type definitions can be scalar or complex, currently based on the XML Schema (XSD).
- Message – an abstract, typed definition of the data being communicated.
- Operation – an abstract description of an action supported by the Web Service.
- Port Type – an abstract set of operations supported by one or more end points.
- Binding – a concrete protocol and data format specification for a particular port type.
- Port – a single endpoint that is an instantiation of a port type in combination with a binding and a network address.
- Service – a collection of related ports.

As listed above, a Web Service is defined in a collection of ports which, in turn, are a collection of abstract operations and messages bound to a concrete protocol and data format specification. Keeping the operations and messages abstract allow them to be bound to different protocols and data formats such as SOAP, HTTP GET/POST or MIME.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
       a2bML Framework 1.0
       A2B Event Service: message and port type definitions
       Copyright 2000 Questra Corporation
-->

<definitions name="EventService"
       targetNamespace= Deleted for Clarity>

       <!—Section 1 -- Datatypes -->
       <types>

          <xsd:element name="EventInfoItem">
              <xsd:complexType content="elementOnly">
                 <xsd:annotation>
```

```
                        <xsd:appinfo> Holds event description.</xsd:appinfo>
                   </xsd:annotation>
                   <xsd:group>
                        <xsd:element ref="a2bdt:Name" minOccurs="1" maxOccurs="1"/>
                         <xsd:element ref="a2bdt:Description" minOccurs="1" maxOccurs="1"/>
                   </xsd:group>
                </xsd:complexType>
            </xsd:element>

             <xsd:element name="EventInfoList">
                 <xsd:complexType content="elementOnly">
                     <xsd:annotation>
                          <xsd:appinfo>Container structure for an event.</xsd:appinfo>
                     </xsd:annotation>
                     <xsd:group>
                          <xsd:element ref="EventInfoItem" minOccurs="0"
maxOccurs="unbounded"/>
                     </xsd:group>
                 </xsd:complexType>
             </xsd:element>

             <xsd:complexType name="Event">
                 <xsd:element name="Source" type="xsd:string"/>
                 <xsd:element name="Description" type="xsd:string"/>
                 <xsd:element name="MemberId" type="xsd:string"/>
                 <xsd:element name="GeneratedAt" type="xsd:timeInstant" minOccurs="0"/>
                 <xsd:element ref="EventInfoList" minOccurs="0"
             </xsd:complexType>

        </types>

        <!-- Messages are based on common data type definitions -->
        <import namespace - deleted for clarity -->

        <!—Section 2 -- Message types -->
        <message name="SubmitEventRequest">
            <part name="RequestHeader" element="a2bheader:RequestHeader"/>
            <part name="Event" element="Event"/>
        </message>
        <message name="SubmitEventResponse">
            <part name="ResponseHeader" element="a2bheader:ResponseHeader"/>
        </message>

        <!—Section 3 -- Port types -->
        <portType name="EventPortType">
            <operation name="submitEvent">
                <input message="SubmitEventRequest"/>
                <output message="SubmitEventResponse"/>
                <fault message="a2bdt:Exception"/>
            </operation>
        </portType>

        <!—Section 4 -- Soap Binding -->
        <binding name="EventSoapBinding" type="EventPortType">
             <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="submitEvent">
                <soap:operation soapAction="http://a2b.example.questra.com/EventService"/>
                    <input>
```

```
                      <soap:header/>
                      <soap:body use="encoded"
                          namespace="http://a2b.example.questra.com/event.xsd"
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
                   </input>
                   <output>
                      <soap:header/>
                      <soap:body use="encoded"
                          namespace="http://a2b.example.questra.com/event.xsd"
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
                   </output>
            </operation>
        </binding>

        <!—Section 5 -- Service -- A Combination of a port and a binding -->
        <service name="EventService">
            <port name="EventSoapPort" binding="EventSoapBinding">
               <soap:address location="http://a2b.example.questra.com/EventSoapService"/>
            </port>
        </service>

</definitions>
```

Figure 5 WSDL Document

Figure 5 shows an example of a WSDL document. It describes an Event Service that might be invoked by some remote device wanting to indicate a fault condition. Section 1 of the document describes the data types used by the event service. Of particular interest is the complex type “Event” that contains information useful to the processing of the fault including the ID of the device submitting the event, a description of the event and a timestamp of when the event occurred. Section 2 of the WSDL document describes the messages that are involved in invoking a service. In this simplified case there are two messages – one to actually invoke the service (SubmitEventRequest) and a response message sent from the service to the requestor (SubmitEventResponse). Section 3 of the document describes the abstract ports available. This example has one port that supports one operation. The operation is accessed through a SubmitEventRequest message, when the operation completes it sends out a SubmitEventResponse message. If an error occurs during processing the operation sends out a a2bdt:Exception message which is detailed in separate data type definition document. Section 4 of the document defines an abstract binding of an operation (SubmitEvent) to a SOAP remote procedure call (RPC) mechanism using HTTP as the transport mechanism. Both the input and output messages of this operation are SOAP encoded. Finally, Section 5 of the document ties everything together to create a concrete service. What makes it real is the inclusion of a location where the service can be accessed, in this case it is a URL.

While this looks like a complex document it is no more intimidating than some of the API documents that exist for making operating system calls or accessing functions from a software library. There are also code generation tools targeted for embedded systems that can read in a WSDL document and output skeleton interfaces and classes for a variety of embedded programming languages including C and C++. These code modules enable transparent access to a Web Service through a local proxy mechanism. Accessing a Web Service then becomes as simple as making a local function call.

UDDI

The final, and optional, piece in the Web Services stack in the Universal Description, Discovery and Integration (UDDI) specification. UDDI defines a way to publish information about Web Services as well as providing a mechanism to discover what Web Services are available. At its bare essence, UDDI is a registration system instantiated as an series of XML files and associated schema that contain a description of a business entity and the Web Services that it offers. It is envisioned that there will be many public UDDI registration servers distributed about the Web that continually replicate data amongst themselves.

The UDDI specification provides a programmatic interface that allows a business to register a Web Service as well as search through the registry for a specific Web Service. Once the desired Web Service is identified a pointer to the location of the WSDL document is provided. It is important to note that UDDI is entirely optional. Companies with Web Services that want to limit specific functionality to people or devices of their choosing need not advertise their service externally.

Web Services in Embedded Systems

With an introductory understanding of Web Services under our belts we can now focus our attention on the question that many embedded developers must be asking –"This is all Information Technology (IT) related stuff, why should I care?" To answer that question all we need to do is revisit the beginning paragraphs of this article. The lines between embedded device and enterprise software are blurring rapidly. Intelligent devices in the field contain incredibly valuable data about status, historical usage, consumables needs, wear and tear and other parameters. In the continuing evolution of the connected device (Figure 6),

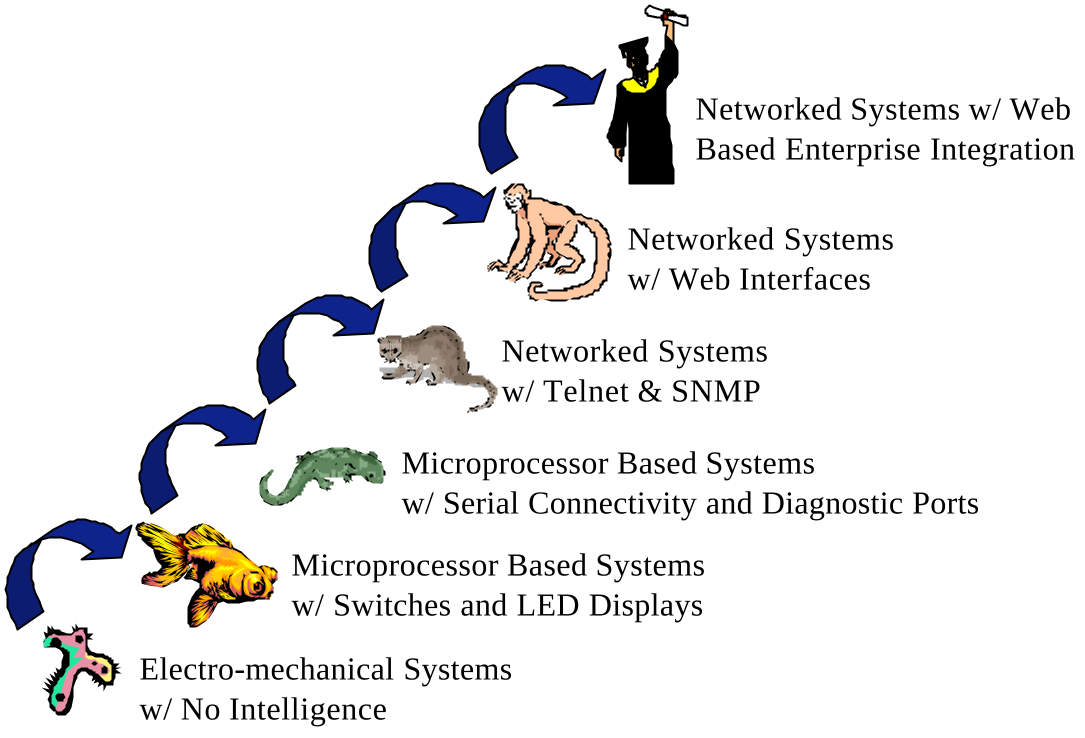
Networked Systems w/ Web Based Enterprise Integration
Networked Systems w/ Web Interfaces
Networked Systems w/ Telnet & SNMP
Microprocessor Based Systems w/ Serial Connectivity and Diagnostic Ports
Microprocessor Based Systems w/ Switches and LED Displays
Electro-mechanical Systems w/ No Intelligence

Figure 6 The Evolution of Connected Devices

devices have moved from standalone systems to the many Internet enabled devices that exist today. However most developers have stopped short of full evolution by providing simple Web interfaces or, at best, point solutions. The drawback of these systems is that they don't get the information into the enterprise systems that can extract the tremendous value that exists. That value may expose itself as reduced field service operating costs, usage based billing, additional revenues through e-replenishment systems or some other value added service that can be provided through the device. In order to do this, it is essential that an end-to-end solution connect the devices into enterprise applications such as an e-commerce, ERP or Field Service application. What better way to accomplish this goal than by speaking what is becoming the native language of these enterprise applications – namely Web Services. A couple of concrete examples should help clarify how an embedded device might use a Web Service.

Accessing Web Services

A digital copier has detected a failure condition in on of its motors. In the prehistoric era of intelligent devices, the best the copier might do is write to a log file and put a message on its LCD interface to indicate that service is required. In a Web Services world, the device may execute the previous two steps, but also call what appears to be a local Event Notification function. Under the hood, a Web Service proxy would take the notification, wrap it up in a SOAP encoded message and invoke a remote Event Web Service. The Event Web Service would take this event, a motor failure, and process it according to its pre-defined workflow and business rules. The result of that processing may be the invocation of other Web Services, such as submission of a Service Request into a field service application as well as a check on the availability of a spare motor from an inventory management system. Through this use of Web Services and Enterprise Application Integration, this simple request may trigger a variety of actions within the enterprise – enabling a highly automated environment. The result is that a simple component failure has automatically set into motion a complex series of events (before the customer may even be aware of the problem) that ultimately results in a faster response to a problem and a high degree of customer satisfaction. This is just one example of a Web Service that could provide value to an embedded system. Ordering consumables, usage metering, operational efficiency monitoring are just some of the functions that could be provided with the proper Web Services.

Providing Web Services

Outbound invocation of a Web Service makes sense, but what about the other direction? Why would a device want to be the provider of a Web Service? To continue the previous example, let's say that the field service application has sent a wireless notification to a field technician's PDA. In the old days, the technician would call the customer to schedule a visit, arrive at the site and plug in to the copier to perform a set of diagnostic routines. All too often this diagnostic sequence suggested that the proper fix involved a part that the technician did not have with him and a trip to a warehouse was in order. In a Web Services world, the technician could use his PDA to remotely invoke the diagnostic routines; in essence the PDA is the client to the Web Servers offered up by the device. The technician can perform the diagnosis remotely and arrive at the customer site with the proper replacement parts in hand. Other functions such as firmware upgrades and remote control could also be provided using the Web Services standards.

Furthermore, by allowing remote devices to "publish" their available services to public or private UDDI registries, the availability of services such as Remote Diagnostics or Firmware Upgrades can be determined dynamically. This provides greater flexibility and extensibility when creating applications that a typical Field Service technician would use. No longer does this functionality have to be hard coded for specific device types.

Design Issues

Each new technology seems to bring with it a whole new set of design tradeoffs and issues. Web Services are no different. Embedded systems just seem to exacerbate the problem. Limited resources, severe cost constraints, and operational considerations all combine to create a complex set of engineering tradeoffs. A discussion of some of the purported issues surrounding Web Services follows.

Security

Security is a major issue anytime you are exposing device information to a public network. Security has many aspects to it including encryption, authentication and authorization. Unfortunately, the SOAP specification is noticeably silent about this issue. Any serious implementation of Web Services must address such issues. Luckily SOAP allows for an XML wrapper to be placed around the actual SOAP message. Within that message framework credentials can be presented and used to authorize a specific action. Fortunately, some Web Service framework implementations targeted at the embedded world do support authenticated access to authorized services. This works both ways in that only authorized people can access certain Web Services provided by a device, while certain Web Services provided by an enterprise may only be available to authorized devices.

Complexity

Just as when Object Oriented programming (OOP) started to filter down into embedded systems, detractors may say that while Web Services are fine for large systems, it just doesn't make sense for the embedded world. Just as with OOP and C++, people also tend to confuse the concept with a specific implementation. Web Services are no different – they combine some of the best features of OOP and distributed computing. Let the processor best suited to a particular process or calculation handle that task but keep the actual inner workings of that task hidden behind a specific interface. Implementing the networking, XML parsing and SOAP encapsulation can be a bit intimidating but luckily there are tools designed to abstract away much of the complexity. For example, one company provides a WSDL compiler targeted specifically for embedded systems. Such a code generator takes an XML based WSDL document as an input and produces C++ code that implements a proxy for a Web Service. For the application developer this means that the Event Service (highlighted in the previous examples) is accessed with what appears to be a simple local function call such as *Event.SubmitEvent(MotorFailure*). The proxy software takes that function call, wraps it in a SOAP message and sends it out over the network. When the response to the SOAP message arrives, it is parsed and the result code is returned to the calling routine. This is especially attractive when taking an existing design and giving it the capability to access Web Services. Most existing devices write to a log file or output a diagnostic code when a fault is detected. Submitting that fault event to an enterprise application becomes a matter of adding one more function call in the fault handling routine.

Resource Constraints

It seems that one of the main features that define an embedded system is that you don't have enough memory, processing power or some other resource to do what you want to. The thought of adding an XML parser and SOAP encoding engine to a system seems problematic at best. In many cases that might be true – a full-blown XML parser can easily add over 180Kbytes of code. Fortunately, many of the features of the XML standard is not required for SOAP encoding. A well pared XML parser that fully supports SOAP can be under 20 Kbytes in size. For those truly constrained devices that still can't even handle that small amount of code, Web Services can also be invoked without using SOAP at all. The WSDL specification also allows a port to be bound to an HTTP POST or GET verb targeted at a specific address. This allows the invocation of a Web Service be a simple as sending an HTTP POST or GET URL-encoded request to a specific URL as shown below:

```
POST /EventService/EventServlet HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: nnnn
[CR][LF]
operation=EventService&authenticationToken=xXXdDj2edph
&description=motot+failure&source=54321
```

Figure 7 URL Encoding

Web Service frameworks targeted towards embedded systems exist that provide an HTTP client & server and support URL encoding while consuming under 5Kbytes of code space.

Verbosity

The verbosity of the text oriented HTTP solution has multiple system impacts, affecting RAM usage, bandwidth requirements and operating costs. XML is a text based language and provides the significant advantage of making platform independence a trivial task. The downside is that text based systems are inherently less efficient than a binary system. This leads to more data being transmitted and larger buffers required to both prepare outbound messages and receive inbound messages. Once approach to addressing this downside is a technique called Compact URL encoding. This encoding scheme builds on the URL encoding provision built into the WSDL specification but adds further compression to minimize both RAM usage and the amount of data sent over the network. Savings of 5 to 10x are easily achievable.

8 and 16 Bit Support

Many platforms that claim to support embedded systems really mean they support 32 bit systems that run a real time operating system (RTOS). In other words, they support only about 15% of the processors out in the field. What about the other 85% of the processors out there – are they doomed to proprietary networking system at best or standalone operation at worst? Certain proprietary networking software companies that say an 8 bit processor simply can't handle a full blown TCP/IP stack have been proven wrong (there are at least a half dozen 8 bit TCP/IP solutions available today). Combining these stacks with the URL encoding technique mentioned above creates a whole new opportunity for connected systems. If the view of Web Services as a distributed computing model is subscribed to, many new applications and features are made possible. For example, addressing the slowly deteriorating performance of an industrial compressor can be a complex statistical calculation based on historical data and current environmental factors. If a more capable application server has access to this information it can handle the calculations, generate a service request for a field technician and also provide feedback to the device such as going into a lower rpm mode to minimize vibrations in order to extend the life of the asset until help arrives. This gives the end customer continued use of that asset but perhaps at a lower capacity. Eight bit applications that can invoke two different Web Services over a TCP/IP stack (with full gateway routing support) have been demonstrated in under 20Kbytes of code on an 8051 type architecture. Consider the impact of even the most mundane devices such as a compressor or smart sensor being able to participate in the community of a business enterprise.

Enterprise Integration

The large number of packaged and custom developed enterprise applications is only surpassed by the incredible number of different types of embedded systems. The business rules surrounding specific events and actions add even more complexity. In order to really extract the value of the information provided by these newly connected devices it is critically important that this information is accessible to the enterprise applications that can analyze the information and apply the business rules. Standalone visualization tools, such as an HP OpenView™ or Micromuse NetCool™ really only provide a view at the current point in time, historical information, which is extremely valuable, is often lost. Other solutions that provide a database to collect historical information along with a specific point application are also available. Unfortunately, these solutions provide little or no integration into the existing enterprise solutions that often have cost a company several million dollars. The different data models and APIs make this integration an onerous task. In addition, devices may want to talk to multiple applications, such as both a Field Service application and a spare parts procurement system making things even more complicated. By incorporating customizable workflow engines into an enterprises Web Service environment, end-to-end solutions can be created that benefit all aspects of both the service and supply chains. As mentioned previously, enterprise applications are quickly moving to the Web Services model. What better way to ease the integration of devices into this enterprise world than to speak the language.

Summary

Web Services are rapidly becoming the native language of business applications. History shows that technologies designed for the enterprise often find their way into embedded systems and that the speed of adoption for these technologies is increasing. Capturing the real value of connecting devices to the Internet goes much further than providing a standalone database for collecting information or providing some proprietary XML interface.

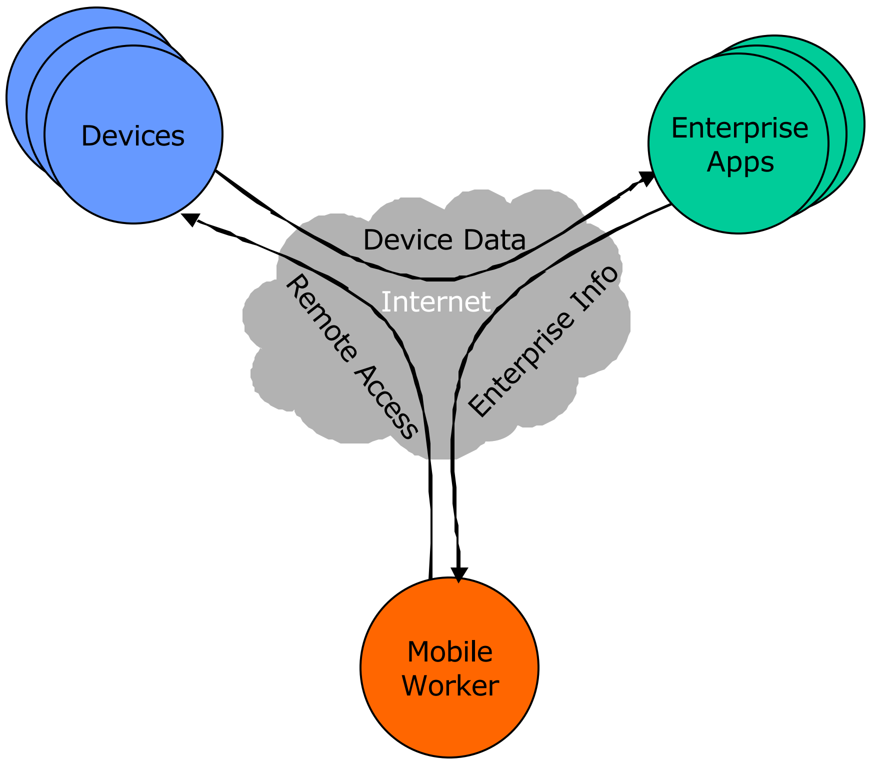

Figure 8 Integrating People, Devices and Enterprise Systems

As shown in Figure 8, a real system level approach results in the connection of the devices that contribute information, the enterprise systems that can analyze the information and apply the proper business rules, and the increasingly mobile workers who can act on the results of the aforementioned analysis as well as remotely interact with the devices. It is only when these three constituencies are served that the tremendous value of the connected world can be completely realized.

References

1 "A2B Technology Overview", Questra Corporation, June 2001, http://www.questra.com
2 "SOAP Specification V1.1", W3C Note 08, May 2000, http://www.w3.org/SOAP
3 "WSDL Specification", W3C Note 15, March 2001, http://www.w3.org/TR/wsdl
4 "UDDI Specification", uddi.org, September 2000, http://www.uddi.org

Questra Corporation
350 Linden Oaks
Rochester, New York 14625
Phone: 716.381.0260
Fax: 716.381.8098
http://a2b.questra.com

Questra is the leading software and services firm focused on architecting, building and deploying Appliance-to-Business™ (A2B™) solutions. A2B solutions power universal connectivity between enterprise systems and remote devices, enabling companies to create new revenue channels, reduce costs, increase business efficiencies and build stronger customer relationships. Questra A2B solutions are built on an open, scalable application platform that provides the critical services needed to connect any smart device with enterprise CRM, eBusiness, ERP and legacy systems. Questra provides comprehensive platform customization and integration services to deliver end-to-end A2B solutions for sales, service and commerce. For more on Questra, visit a2b.questra.com, or contact us at info@questra.com, 800.785.6359.