An Embeddable Standards Compliant Web Service Server

Terry H. Ess
T. H. E. Solution LLC

INTRODUCTION

The quote that by 200x there will be more smart devices on the internet then people has become widely accepted. There are a number of TCP/IP high level protocols that have good potential for connecting smart devices depending on the application and platform. One of the most general (and complex) is the evolving series of transport and content protocols that are bundled under the banner web service. At its essence this is nothing more than an interface that transports XML documents using the HTTP protocol. At this basic level a fairly general purpose server for information retrieval can be implement in less than 500 lines of C (this represents approximately 3 KB of ROM in a M68HC12 micro-controller; download www.the-solution-llc.com/webservice.zip for all the technical details). Use of the service can be easily incorporated into a web page (or other media) with just a small amount of scripting (VB or Java). However, if we want our device to tie to the IT infrastructure or we want capabilities beyond just information retrieval, then we need to go further. In the IT world we could use Web Servers extended with .NET or J2EE "middleware". For many smart devices this is not an acceptable answer. Not only are these components large they are also maintenance intense. So what would it take to make a fairly general purpose server that would meet our expanded requirements and still be embeddable? This article will explore exactly that.

REQUIREMENTS

The design of the reference implementations was driven by five primary requirements;

1. Load characteristics assumptions.
2. The desire to support the general purpose application level communications required for remote command and control.
3. The need to provide an interface that meets a "web service standard" likely to be acceptable for use by an IT organization.
4. The desire to make an implementation that can be easily ported to modest platforms. This includes a small footprint and a very limited set of platform resource dependencies.
5. The need to provide security that is sustainable on modest platforms.

First, the design is driven by some load characteristic assumptions:

1. There is a relatively low level of traffic (on the order of a ten "messages" per second).
2. The device does not need to support a large number of simultaneous connections.

We want our interface to provide a very general purpose capability so that it will meet all our information retrieval, configuration and control needs. The most general means of communications is to combine a command with a response, a conversation. A message is

the basic unit of communication to command an action or respond to a command. At the minimum it is composed of a function code. Typical functions supported are:

1. START – start the indicated action.
2. STOP – stop the indicated action.
3. SET – set a parameter.
4. GET – get a parameter.

Additional message fields are required based on the nature of the message's function. A summary of a typical command and response message set is provided in the following table:

Command	Response
START,action	START,action,status
STOP,action	STOP,action,status
SET,parameter id,parameter modifier(optional),parameter value	SET,parameter id,status
GET parameter id,parameter modified (optional)	GET,parameter id,parameter modifier (if present in command),status, parameter value

For the reference application we will limit ourselves to two actions, locking the device (i.e. preventing the modification of parameters or the execution of actions by anyone else) and "operating" and four parameters, all short (two bytes).

We need to meet "web service standards" at the interface (not necessarily internally). This is a moving target but there is a good practical starting point, the standards required by both the .NET and Java based tools that automate the construction of web service client stubs (this is pretty much the WS-I Basic Profile 1.0 without UDDI):

1. XML 1.0
2. XML Schema
3. HTTP 1.1
4. SOAP 1.1
5. WSDL 1.1

Since we control the definition of the web service, we will make some definition decisions that help to simplify our life (and appear to be the direction of the future gauging by the drafts of SOAP 1.2):

1. The XML will use the document style.
2. The encoding in the XML documents will be "literal" (i.e. they use XML Schema based definitions not SOAP).

The reference implementation's dependency on platform provided resources was kept to:

1. A small sub-set of the standard C run time library that are available for most micro-controllers: memcmp, strcpy, strncpy, strcat, strncat, sprintf, strlen, strncmp and atoi.
2. A TCP/IP socket interface provided by either a software stack or by an internet chip (iChip from One Connection).
3. Basic multi-tasking (threading) capabilities.

Finally, a "basic" level of security is assumed to be required. The scheme used in the reference implementation is based on the concept of a one time pad and is suitable for even "deeply" embedded applications. The specific parameters used for the reference implementations follow:

1. The TCP/IP socket session will be the level at which access control is exercised.
2. Access security is provided by user authentication and different authorization levels matched to the command set (get data, set data, start/top actions).
3. At the beginning of each session the user must authenticate itself using a connection message.
4. The connection message has an encoded section which includes the OR of the user's password and the current key .
5. The response to the connection message includes the exclusive OR of the current key and the new key. The new key is generated using a random number algorithm that has a good distribution and is suitable for use in generating cryptographic key material.

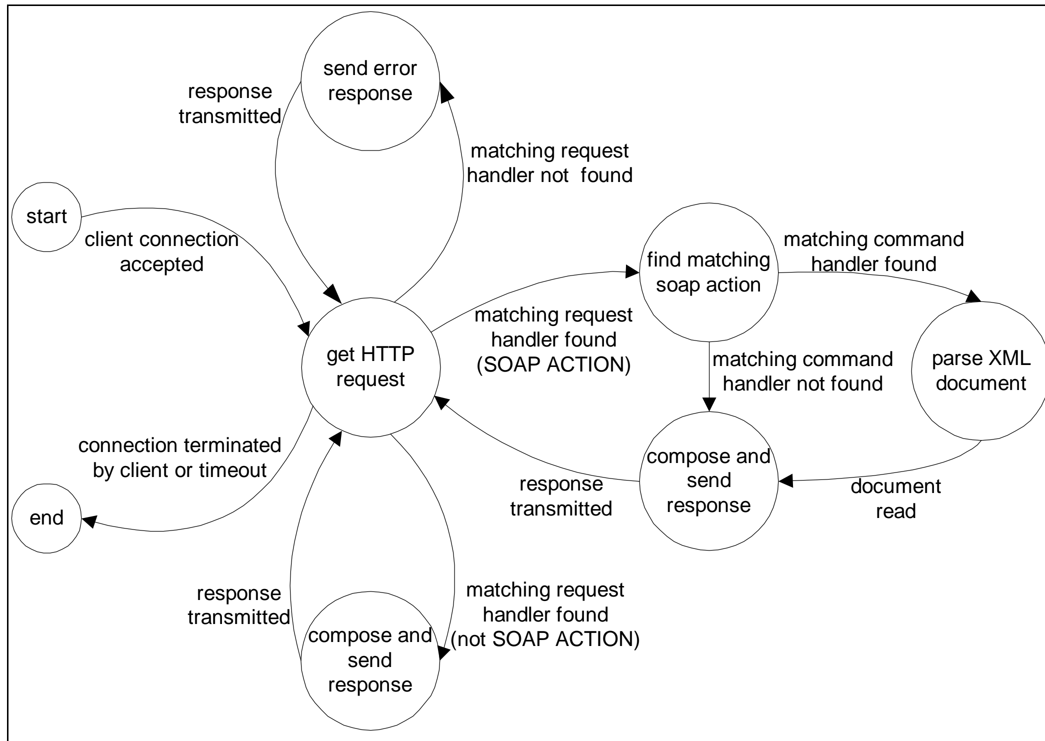
DESIGN

The external world knows the service from a formal definition, the WSDL document. This can be a fairly lengthy document so luckily this is produced as part of the design process (in fact it makes a really good starting point for the design) and not by the service itself. The WSDL for the reference implementation is included as an appendix. In the case of the reference implementation we provide the ability to retrieve a copy of this definition from the device using any web browser.

The overall sequence of the application is straightforward:

1. Open a TCP server socket.
2. Wait for a client connection.
3. For each client connection start a new thread to handle the client's requests (NOTE: The thread per client model is fine for cases where a relatively small number of simultaneous connections is maintained. In the commercial Web server arena where the number of connections is in the thousands this does not work well. Then you have to do things like thread pooling).
4. The client handling thread waits for a complete HTTP request and based on the request provides the appropriate response.
5. When the client closes the socket or the socket hits a inactivity timeout, the client handling thread closes its side and terminates.

The workhorse in the code is the client handling. It can be described very succinctly as a state machine:



For each request it tries to find a handler for the specific request. A typical GetShort request follows:

```
POST /DeviceIO HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; MS Web Services Client
Protocol 1.1.4322.573)
Content-Type: text/xml; charset=utf-8
SOAPAction: "DeviceIO#GetShort"
Content-Length: 313

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetShort xmlns="http://device/def/">
<id xmlns="">1</id>
</GetShort>
</soap:Body>
</soap:Envelope>
```

Finding a handler makes use of a lookup table that contains a list of accepted requests (the first line in the request) and a pointer to the function that will handle this request. In the reference implementation this looks like:

```
struct request_handler
```

```
        {
        char *name;
        void (*function)(class client_handler *,char *);
        };

struct request_handler client_handler::request_handlers[] =
        {
        {"GET ",GetHandler},
        {"POST /DeviceIO",SoapHandler},
        {"",NULL}
        };
```

The subsequent handling of soap requests by "SoapHandler" is done in a similar manner using the SOAPAction line. In the reference implementation the look up table looks like:

```
struct soap_handler
        {
        char *name;
        void (*function)(class client_handler *);
        unsigned char required_auth;
        };

struct soap_handler client_handler::soap_handlers[] =
        {
        {"SOAPAction: \"DeviceIO#Connect\"",MakeConnect,AUTHORIZED_NONE},
        {"SOAPAction: \"DeviceIO#GetShort\"",GetShort,AUTHORIZED_GET},
        {"SOAPAction: \"DeviceIO#SetShort\"",SetShort,AUTHORIZED_SET},
        {"SOAPAction: \"DeviceIO#Start\"",StartAction,AUTHORIZED_ACTION},
        {"SOAPAction: \"DeviceIO#Stop\"",StopAction,AUTHORIZED_ACTION},
        {"",NULL}
        };
```

How we handle the parsing of XML documents is the most critical item in terms of meeting a small footprint requirement. The typical XML parser whether of the DOM or SAX variety is large. Since we have control of the XML layout we can take advantage of this information to make a minimal, sequential parser. Using a list of tags and associated handlers, the parser does nothing more then find each tag, parse the "value" as a string and invoke the associated function which knows how to convert, store etc. the item. In the reference implementation the sequence list for a SetShort XML document is:

```
struct xml_element_handler
        {
        struct *tag;
        void (*function)(class client_handler *,char *);
        };

struct xml_element_handler client_handler::set_short[] =
        {
                {"<id",SetShortId},
                {"<value",SetShortValue},
                {"",NULL}
        };
```

Once the parsing is completed the saved values are checked and the appropriate response composed and sent as the response to the original HTTP request. The response for a successful GetShort follows:

```
HTTP/1.1 200 OK
MIME-version: 1.0
Content-type: text/xml
Connection: keep-alive
Content-length: 276

<?xml version="1.0" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetShortResp xmlns="http://device/def/">
<id xmlns="">1</id>
<value xmlns="">1</value>
</GetShortResp>
</soap:Body>
</soap:Envelope>
```

RESULTS

A server reference implementation for both a software based TCP/IP stack and a iChip based socket interface were developed as Windows console applications. Both of the implementations were tested against client programs written in C# .NET (Visual Studio 2003, .NET Framework 1.1) and Java (Eclipse, J2SE 1.4.2, Sun's Java Web Services Developer Pack 1.3). In both cases access to the web service was automatically generated from the WSDL and the client program simply called stubs. The software stack version of the server required less then 1000 lines of C++. The chip based version required less then 1400 lines of C++ [NOTE: since this implementation requires no software based TCP/IP stack the overall size is significantly smaller]. This would be in the range of 6 to 9 Kbytes of ROM in a M68HC12 micro-controller. The addition of each new message to the web service would require on the average only fifteen lines of code, so even a very complex message set would not require significantly more code. Though this is not tiny it can easily fit in a wide variety of devices implemented on modest platforms.

The performance of the internet chip version was substantially less then that of the socket interface version. Part of this is undoubtedly caused by the fact that the chip set used only supported a dial-up connection. But the main contributor is the serial interface used between the internet chip and the main processor and the specific procedures that needed to be completed to retrieve data received by the internet chip. In general the internet chip should be restricted to five or less simultaneous connections (it supports up to ten). In many if not most cases neither the performance or number of simultaneous connections is not likely to be an issue.

There is of course a significant price for using the web service approach, the amount of data moved is significantly larger. Where a proprietary protocol running on top of TCP/IP could use less then ten bytes to perform a GetShort, the web service used 900

bytes. The reference implementations includes the ability to limit the number of simultaneous connections to keep the load this represents from overwhelming the device.

Security is probably the biggest problem in this connected world. Currently there is no standardized method for web service security, though one is in the works. The current draft does not appear to be suitable for smart devices, at least ones implemented on a modest platform. The scheme used in the reference implementations provides good but not perfect security. If sessions are maintained for long periods, especially if the device's IP address is static and exposed to the outside world, there is some chance of a session being hijacked. The reference implementations addresses this partially by closing a connection that has been inactive for some time. The scheme provides no protection against monitoring or denial of service type attacks. The scheme does impose some potential administrative problems on the client side since each connection requires the key generated in the prior connection for the specific user.

Appendix A, Reference Links

WS-I Basic Profile
Simple Object Access Protocol (SOAP) 1.1.
Extensible Markup Language (XML) 1.0 (Second Edition).
RFC2616: Hypertext Transfer Protocol -- HTTP/1.1.
Web Services Description Language (WSDL) 1.1.
XML Schema Part 1: Structures.
XML Schema Part 2: Datatypes.

Appendix B, Server Reference Implementation WSDL

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions
   targetNamespace="http://device/def/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://device/def/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <types>
      <xs:schema targetNamespace="http://device/def/" xmlns="http://device/def/">

         <xs:complexType name="Passcode">
            <xs:sequence>
                <xs:element name="data" type="xs:unsignedByte" minOccurs="16"
maxOccurs="16"/>
                </xs:sequence>
         </xs:complexType>

         <xs:complexType name="ConnectParameters">
            <xs:sequence>
               <xs:element name="user" type="xs:string"/>
               <xs:element name="pass" type="Passcode"/>
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="ConnectResponse">
            <xs:sequence>
               <xs:element name="result" type="xs:boolean"/>
               <xs:element name="pass" type="Passcode"/>
            </xs:sequence>
         </xs:complexType>
         <xs:element name="Connect" type="ConnectParameters"/>
         <xs:element name="ConnectResp" type="ConnectResponse"/>

         <xs:complexType name="GetParameters">
            <xs:sequence>
               <xs:element name="id" type="xs:short"/>
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="GetShortResponse">
            <xs:sequence>
               <xs:element name="id" type="xs:short"/>
               <xs:element name="value" type="xs:short"/>
            </xs:sequence>
         </xs:complexType>
```

```
        <xs:element name="GetShort" type="GetParameters"/>
        <xs:element name="GetShortResp" type="GetShortResponse"/>


        <xs:complexType name="SetShortParameters">
          <xs:sequence>
            <xs:element name="id" type="xs:short"/>
            <xs:element name="value" type="xs:short"/>
          </xs:sequence>
        </xs:complexType>
        <xs:complexType name="SetResponse">
          <xs:sequence>
            <xs:element name="id" type="xs:short"/>
            <xs:element name="result" type="xs:boolean"/>
          </xs:sequence>
        </xs:complexType>
        <xs:element name="SetShort" type="SetShortParameters"/>
        <xs:element name="SetShortResp" type="SetResponse"/>

        <xs:complexType name="StartStopParameters">
          <xs:sequence>
            <xs:element name="op_code" type="xs:short"/>
            <xs:element name="param" type="xs:short"/>
          </xs:sequence>
        </xs:complexType>
        <xs:complexType name="StartStopResponse">
          <xs:sequence>
            <xs:element name="op_code" type="xs:short"/>
            <xs:element name="result" type="xs:boolean"/>
          </xs:sequence>
        </xs:complexType>
        <xs:element name="Start" type="StartStopParameters"/>
        <xs:element name="StartResp" type="StartStopResponse"/>
        <xs:element name="Stop" type="StartStopParameters"/>
        <xs:element name="StopResp" type="StartStopResponse"/>

      </xs:schema>
    </types>
    <message name="ConnectResponseMessage">
      <part element="tns:ConnectResp" name="parameters"/>
    </message>
    <message name="ConnectMessage">
      <part element="tns:Connect" name="parameters"/>
    </message>

    <message name="GetShortResponseMessage">
```

```
    <part element="tns:GetShortResp" name="parameters"/>
</message>
<message name="GetShortMessage">
    <part element="tns:GetShort" name="parameters"/>
</message>

<message name="SetShortResponseMessage">
    <part element="tns:SetShortResp" name="parameters"/>
</message>
<message name="SetShortMessage">
    <part element="tns:SetShort" name="parameters"/>
</message>

<message name="StartResponseMessage">
    <part element="tns:StartResp" name="parameters"/>
</message>
<message name="StartMessage">
    <part element="tns:Start" name="parameters"/>
</message>

<message name="StopResponseMessage">
    <part element="tns:StopResp" name="parameters"/>
</message>
<message name="StopMessage">
    <part element="tns:Stop" name="parameters"/>
</message>

<portType name="DeviceIO">
    <operation name="Connect">
        <input message="tns:ConnectMessage"/>
        <output message="tns:ConnectResponseMessage"/>
    </operation>

    <operation name="GetShort">
        <input message="tns:GetShortMessage"/>
        <output message="tns:GetShortResponseMessage"/>
    </operation>

    <operation name="SetShort">
        <input message="tns:SetShortMessage"/>
        <output message="tns:SetShortResponseMessage"/>
    </operation>

    <operation name="StartCmd">
        <input message="tns:StartMessage"/>
        <output message="tns:StartResponseMessage"/>
```

```
        </operation>

        <operation name="StopCmd">
            <input message="tns:StopMessage"/>
            <output message="tns:StopResponseMessage"/>
        </operation>

    </portType>
    <binding name="DeviceIOBinding" type="tns:DeviceIO">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="Connect">
            <soap:operation soapAction="DeviceIO#Connect"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>

        <operation name="GetShort">
            <soap:operation soapAction="DeviceIO#GetShort"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>

        <operation name="SetShort">
            <soap:operation soapAction="DeviceIO#SetShort"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>

        <operation name="StartCmd">
            <soap:operation soapAction="DeviceIO#Start"/>
            <input>
                <soap:body use="literal"/>
            </input>
```

```
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>

    <operation name="StopCmd">
      <soap:operation soapAction="DeviceIO#Stop"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>

  </binding>
  <service name="Service">
    <port binding="tns:DeviceIOBinding" name="DevicePort">
      <soap:address location="http://localhost:2000/DeviceIO"/>
    </port>
  </service>
</definitions>
```