

# Accessing Devices Using a Web Service

Terry H. Ess

The time when embedded devices and factory floor machines had only minimal interaction with humans is gone. Today the ability to access a device from anywhere is expected. This is a significant challenge when we are heavily constrained by hardware size and costs. However, with commonly available software components and a little knowledge, we can incorporate a “universal” access interface for data gathering/surveillance that maintains a small footprint and is secure.

## 1. INTRODUCTION

Today the ability to access an embedded device or factory floor machine from anywhere is expected. There are at least four questions we need to address to examine the issue of universal access:

1. Who needs access?
2. What will they do?
3. How will access be provided?
4. When will access be allowed?

In this paper we will restrict our interest to an open technology approach for providing a “read only” channel. Though restricted in nature this often meets the primary access requirements of both customers and manufacturers, answering much of our four questions. Specifically we will:

1. Use the most ubiquitous and cost effective transport possible, TCP/IP on top of an appropriate physical layer (10 BaseT Ethernet, analog modem, direct serial connection etc.)
2. Employ two open interoperable technologies, HTTP and XML, to provide our top-level control and information transfer.

In the current parlance, we are providing a basic web service. The normal means of achieving this capability is by using a full function Web Server with extensions to provide an interactive exchange between the client and the server, e.g. Microsoft’s IIS with ASP, or Macromedia’s ColdFusion with JSP etc. This is a complicated solution that can prove to be costly and difficult to administer and secure. For an embedded application where every square inch of board space and dollar of hardware is important, it is often just not practical.

Luckily there is a straightforward solution that does not require the hardware resources of a commercial web server. Even a rather cursory knowledge of HTTP and XML enables a software engineer to develop a secure application specific solution with the smallest possible resource demands. The basic facts are:

1. The transport for HTTP is a TCP/IP socket that uses TCP port 80 on the server.
2. HTTP is a simple text based protocol, most of which we can ignore.
3. XML is a text based mark-up language that is easy to understand and create. It does not have to be any more complicated than we want it to be.

We need a socket interface to a TCP/IP stack, a relatively small piece of software, the Web Service Interface, and we are there. We will travel down the road of developing this interface one step at a time.

## 2. UNDERSTANDING HTTP

The Hypertext Transport Protocol provides a simple conversation oriented exchange between computers. An HTTP based client application, e.g. a Web browser, establishes a TCP/IP socket connection with an HTTP server and then carries on a request – response conversation.

### 2.1. HTTP Requests

A typical HTTP request looks like:

```
GET /listener.htm HTTP/1.1<cr><lf>
Accept: image/gif, application/vnd.ms-powerpoint, application/vnd.ms<cr><lf>
Accept-Language: en-us<cr><lf>
Accept-Encoding: gzip, deflate<cr><lf>
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)<cr><lf>
Host: the4<cr><lf>
Connection: Keep-Alive<cr><lf>
<cr><lf>
```

The key things to note are:

1. An HTTP request is just a series of text lines (i.e. displayable text terminated with a carriage return and line feed). An empty line terminates the request.
2. It provides a wealth of information, most of which we can ignore.
3. What is important for us is the first line. This includes three key pieces of information. The action requested, the item associated with this request and the version of HTTP used.

The current version of HTTP, 1.1, supports a number of different actions, POST, PUT, COPY etc. but the only one we need for providing information is GET. A side effect of only supporting GET is that the actions that are prone to be used by hackers will not be supported.

## 2.2. HTTP Responses

An HTTP response has one or two parts. The response header provides the status information necessary for the client to make use of the response. If the response includes the movement of data to the client then the second part is the data stream. We will use three types of responses, a generic error response and two different data response. The error response is a one-part (i.e. header only) response and it is:

```
HTTP/1.1 400 Bad Request<cr><lf>
Content-length: 0<cr><lf>
<cr><lf>
```

The key things to note in the error response header are:

1. A response header is just a series of text lines. An empty line terminates the header.
2. The first line in the header provides two basic pieces of information, the HTTP version used (which should match that in the request) and the status of the request.
3. The last line tells the client how big the data stream section will be. Our error response will include no data section.

There are dozens of possible status responses, each composed of a number and a matching text string. For our simple service we only need two, 400 Bad Request and 200 OK. The OK response header is used in the two different data responses we will support. The first of these is our XML response header. This is:

```
HTTP/1.1 200 OK<cr><lf>
MIME-version: 1.0<cr><lf>
Content-type: text/xml<cr><lf>
Connection: keep-alive<cr><lf>
Content-length: 100<cr><lf>
<cr><lf>
```

This header includes a little more information than the error response header. The key things to note are:

1. The overall composition of a response header has not changed.
2. We have included two additional lines which define the nature of the data stream we will be providing:
3. A line that defines the MIME (Multipurpose Internet Mail Extensions) standard version that is being used.
4. A line that defines the content type using the indicated MIME standard, in this case XML.

5. We add the "keep-alive" line so the client will hopefully not drop the socket immediately after each request – response cycle.
6. The last line must contain the actual byte count in the following data stream.

The second data response will allow us to return an HTML page. The only thing new here is the content type:

```
HTTP/1.1 200 OK<cr><lf>
MIME-version: 1.0<cr><lf>
Content-type: text/html<cr><lf>
Connection: keep-alive<cr><lf>
Content-length: 100<cr><lf>
<cr><lf>
```

In general we do not need to support an HTML response. One of the key advantages to providing an XML response is that we are only providing data; we are not specifying how the data should be presented to the user. This keeps what we have to produce relatively small and easy. Picking up this data and placing it into a display application is not difficult since most of the work is done by off-the-shelf XML parsers, but it does require some knowledge on the part of the users. Most of this can be provided in a couple of simple tables and/or an XML style sheet, but it is always good to provide a simple example. Along those lines a single HTML page that includes Visual Basic or Java script can provide a living example of how to access and use the data provided. Then it is up to the user organization to build what ever interfaces it needs.

The second part of the response is the data stream that makes up the response, i.e. we are sending the XML document or HTML page. The only requirement is that the content type and byte count of this data stream be the same as the information provided in the response header.

### 3. UNDERSTANDING XML

The Extensible Markup Language provides a standardized means of describing and transferring data. XML is a meta-markup language, a set of rules for creating semantic tags used to describe data. An XML element is made up of a start tag, an end tag, and data in between. The start and end tags describe the data within the tags, which is considered the value of the element. For example, the following XML element is a <director> element with the value "Ron Howard."

```
<director>Ron Howard</director>
```

The element name "director" allows you to mark up the value "Ron Howard" semantically, so you can differentiate that particular bit of data from another, similar bit of data. For example, there might be another element with the value "Ron Howard."

```
<actor>Ron Howard</actor>
```

Because each element has a different tag name, you can easily tell that one element refers to Ron Howard, the director, while the other refers to Ron Howard, the actor. If there were no way to mark up the data semantically, having two elements with the same value might cause confusion.

In addition, XML tags are case-sensitive, so the following are each a different element.

```
<City> <CITY> <city>
```

An element can optionally contain one or more attributes. An attribute is a name-value pair separated by an equal sign (=).

```
<CITY ZIP="29650">Greer</CITY>
```

In this example, **ZIP="29650"** is an attribute of the <CITY> element. Attributes are used to attach additional, secondary information to an element, usually meta information. Attributes can also accept default values, while elements cannot. Each attribute of an element can be specified only once, but in any order.

A basic XML document is simply an XML element that can, but might not, include nested XML elements. For example, the XML <books> element is a valid XML document:

```
<books>
<book isbn="0-201-000650-2">
<title>The Mythical Man-Month</title>
<author>Frederick P. Brooks, Jr.</author>
</book>
</books>
```

There are some things to remember when constructing a basic XML document:

1. All elements must have matching start and end tags.
2. All elements must be cleanly nested (overlapping elements are not allowed).
3. All attribute values must be enclosed in quotation marks.
4. Each document must have a unique first element, the root node.

#### 4. REQUESTS AND RESPONSES

We now understand the basics of HTTP requests and responses and XML. What we need next is to match requests and responses in a manner that provides the developer with sufficient information to implement the Web service and for a user to access and use the data provided. This is an easy task using two simple tables. For illustration we will use a VERY simple example. In this case the device is a clock. It provides only two pieces of information, the devices software version and the clock's current local time. The tables are:

HTTP Request	Response
GET /Version	Device's software version
GET /Time	Devices' current local time
GET /Listener.htm	The example listener Web page
All other requests	HTTP error response

Response	XML
Software version	<SwVersion> <Major>integer</Major> <Minor>integer</Minor> <Rev>integer </Rev> </SwVersion>"
Time	<Time> <Hour>integer</Hour> <Minute>integer</Minute> <Second>integert</Second> </Time>

*Integer* – signed 32 bit integer number

The HTTP requests are easily made in a Web Browser. For instance the URL to generate the version request would look like "<http://hostname/Version>". If request compatibility with a Web server, e.g. Microsoft IIS using ASP, is important then the request could be changed to a more conventional form such as "GET /Get.Asp?WCI=Version".

## 5. IMPLEMENTATION

We have now covered the background and primary requirement issues of our basic Web service example. Let's take a look at the key parts of implementation. First a couple of key points about this example:

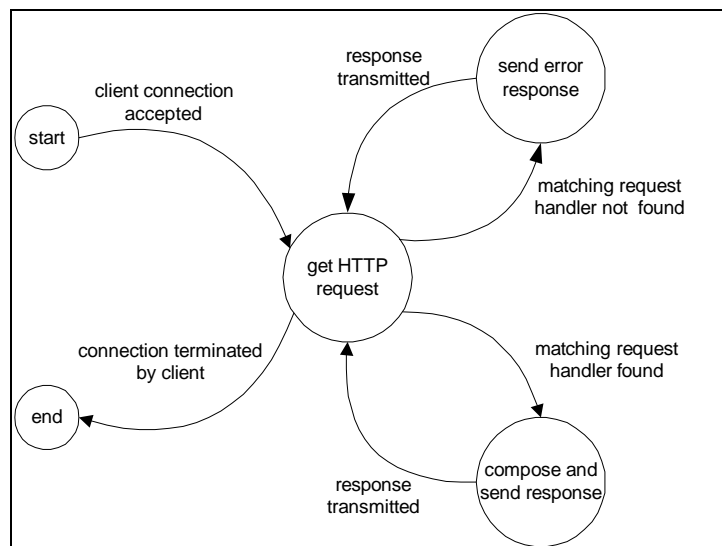
1. The Web service application is a typical "server" type socket application with a minimal console human interface.
2. Key parameters used by the Web service application are provided in a small text configuration file, web.cfg.
3. The examples, Windows, Linux and VxWorks versions, were implemented in C++ using C runtime support as standalone programs. They can be easily ported to straight C where C++ is not supported and/or included as part of an existing piece of software.
4. The primary external facilities required by the Web service application are a socket interface to a TCP/IP communication stack and support for threading.
5. The service client is implemented in the listener HTML page using Visual Basic script and Microsoft's XML parser.
6. The source code for the example Web service application and its consumer can be downloaded from <http://the-solution-llc.com/webservice.zip>. This includes the initial draft of this paper.

### 5.1. Web Service Example Application

The overall sequence of the web service application is straightforward:

1. Open a server type socket on TCP port 80.
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, the client handling thread closes its side and terminates.

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



For each request it copies the HTTP version signature of the request, we need that in the response, and then tries to find a handler for the specific request. Finding a handler makes use of a lookup table that contains a list of accepted requests and a pointer to the function that will handle this request.

```

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

struct request_handler client_handler::request_handlers[] =
{
    {"get /version ",SendVersion},
    {"get /listener.htm ",SendListener},
    {"get /time ",SendTime},
    {"",NULL}
};

```

This simple implementation of matching request and response handlers provides decent scalability and is easily modified to meet any specific need.

An XML response is nothing more than the dynamic composition and transfer of the appropriate XML response “document”. A short code snippet from the Linux version time request handler will illustrate:

```

time_t t;
struct tm *today;
char hr_line[20],min_line[20],sec_line[20];

t = time(NULL);
today = localtime(&t);
sprintf(hr_line,"<Hour>%02d</Hour>",today->tm_hour);
sprintf(min_line,"<Minute>%02d</Minute>",today->tm_min);
sprintf(sec_line,"<Second>%02d</Second>",today->tm_sec);

```

## 5.2.Web Service Client Example

From the client point of view it has made a request and is provided data in the form of a document as the response. Because of the nature of this document, individual items can be easily found and parsed using off-the-shelf software and displayed. The simple listener HTML page, listener.htm, in the example does exactly this using Visual Basic script and the Microsoft XML parser (a COM object). The example could just as easily been implemented in other forms such as a stand alone “web application” using Visual Basic, Java or C++ and any number of XML parsers.

The example client uses the Document Object Model (DOM) parser. The DOM presents an easily processed standardized interpretation of an XML document to applications and scripts that is well suited to relatively small documents. The basic operation of the example is:

- ?? Load the appropriate XML document
- ?? Find each data element required by its tag name.
- ?? Remove the text value of the element.
- ?? Manipulate the text as necessary and display.

The sub-routine that takes care of update the HTML pages time field illustrates just how simple this is:

Sub UpdateTime

```

Dim elemList
Dim xmlDoc
Dim hr_stg
Dim min_stg
Dim sec_stg
dim success

Set xmlDoc = CreateObject("Msxml.DOMDocument")
xmlDoc.async = False
xmlDoc.validateOnParse = False
Err.Clear

```

```

On Error resume next
xmldoc.Load("http://" + web_path + "time")
If xmldoc.documentElement.baseName = "Time" Then
    Set elemlist = xmldoc.getElementsByTagName("Hour")
    hr_stg = elemlist.Item(0).Text
    Set elemlist = xmldoc.getElementsByTagName("Minute")
    min_stg = elemlist.Item(0).Text
    Set elemlist = xmldoc.getElementsByTagName("Second")
    sec_stg = elemlist.Item(0).Text
    WebTime.Value = hr_stg + ":" + min_stg + ":" + sec_stg
Else
    WebTime.Value = "Error"
End If
On Error GoTo 0

End Sub

```

## 6. CONCLUSIONS

With a little bit of knowledge and a relatively small piece of software (the example implementation is only a little over 400 lines of C++) we can provide a basic Web service. Because of the very restricted nature of this implementation, we can provide this service in a very small footprint and with high security while still providing our customers with access to the information they need. If we implement the Web service as a separate process, we can achieve small size, high security and robustness. Such an implementation is provided in T. H. E. Solution's PC on the factory floor toolkit (see [http://www.the-solution-llc.com/development\\_tools.htm](http://www.the-solution-llc.com/development_tools.htm)). Like most engineering solutions, there are limitations. The primary limitations are:

1. This solution is designed to support a relatively small number of connections (in the dozens not hundreds). Because of the limited number of connections supported, client use of proxy web servers can cause a problem. It also makes the Web service vulnerable to denial of service attacks
2. The XML used in this example is extremely simple in nature. Many industry groups are developing specific rules and definitions for data interchange. Specific applications should take advantage of such definitions when they exist.
3. The examples all use a static IP address for the Web service. This may or may not be a good solution for any specific application. The requirement to support dynamic IP addressing can significantly complicate the solution. This is especially true if the application's environment only provides limited support for configuration and naming services such as DHCP and DNS.
4. The most complicated area of the web service concept, and one still very much in progress, is the means to dynamically locate web services and determine the nature of the service. This has not been addressed at all for two reasons. First, in all practical cases this would be handled exterior to the device. Second, it is not at all clear that this type of registry capability is applicable to the problem we are trying to solve.

## BIBLIOGRAPHY

1. World Wide Web Consortium (W3C), Specifications of numerous Web protocols including HTTP and XML, <http://www.w3.org/>
2. Matthew Powell and Leon Brackhnski, "HTTP Revealed", *Microsoft Interactive Developer*, September 1996
3. David Cook, "Write a Simple HTTP-based Server using MFC and Windows Sockets", *Microsoft Systems Journal*, February 1996
5. Microsoft Corporation, XML Tutorial, MSDN Library, July 2001
6. This paper is directly derived from a 2002 IEEE SECon paper of the same name, Copyright 2002 IEEE