XML, SOAP and Web Services in Forth

Stephen Pelc
Microprocessor Engineering
133 Hill Lane
Southampton SO15 5AF
stephen@mpeforth.com

Abstract

Web services enable applications to exchange data using an extension to HTTP. Implementing web services requires extensions to an HTTP server, parsing and generating XML and then interfacing to other applications. This paper discusses what was needed to extend MPE's PowerNet to handle web services, and how we took advantage of Forth itself to simplify the solution.

Introduction

Web services are a a means of exchanging data between applications. The transfers are designed for machine use, not for human use; despite this the transfers are mostly printable. Data is exchanged using XML templates and data descriptions. You can treat XML as an extensible version of HTML with stricter rules.

Unlike many other application interchange protocols such as DCOM, web services are based on open standards and so are not restricted to use under specific operating systems. Because all data transfers are in text form, web services do not suffer from the data marshalling issues of other techniques.This is achieved at the expense of an increase in data size of about 10:1, which can make severe demands on the underlying networks used to link machines. It has long been MPE's opinion that heterogeneous interoperability is most easily achieved using sockets and text transfers. This was an ideal opportunity to test the assertion.

The work was supported by Construction Computer Software (Cape Town, South Africa), and so follows their requirements. The initial design work and test systems were provided by Graham Stevenson of Oxford Network Solutions.

After much reading of documentation and specifications, the implementation order was:

- XML input
- XML output
- Testing against an existing web service
- PowerNet changes
- Generate WSDL files
- Test with Excel

PowerNet v3

PowerNet v3 is a Forth TCP/IP stack with Telnet, web server, CGI and ASP facilities. It has been running for some years on embedded systems. The version for Windows replaces the embedd TCP/IP stack with calls to the Winsock API. Above that level, the multi-threaded

servers require very little change between the embedded and the Windows versions. Scripting facilities are provided by Forth itself.

At a very early stage in the design of PowerNet, we decided to implement each connection to a server as a task, and to treat the TCP/IP sockets as standard Forth I/O streams. Although this can increase the amount of RAM required by a busy server, it has the big advantage of simplicity. An additional advantage is that the usual Forth I/O handling, particularly **KEY EMIT** and friends, can be used with each connection. This design decision was to pay off handsomely when implementing web services.

An example transaction

The following is a SOAP request to a server:

```
POST /service1.asmx HTTP/1.1
Host: oxns.demon.co.uk
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://oxns.demon.co.uk:37851/HelloWorld"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
>
  <soap:Body>
    <HelloWorld xmlns="http://oxns.demon.co.uk:37851/">
      <s1>string</s1>
      <s2>string</s2>
      <i1>int</i1>
    </HelloWorld>
  </soap:Body>
</soap:Envelope>
```

The response is generated from the script file *HelloWorld.aspx*:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
>
  <soap:Body>
    <HelloWorldResponse
      xmlns=http://oxns.demon.co.uk:37851/
    >
      <HelloWorldResult>string</HelloWorldResult>
```

```
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

The request's *SOAPaction* header is parsed to yield *HelloWorld*, which is the required action and corresponds to a Forth wordlist. This name is extended to select the file *HelloWorld.aspx* which is output and processed by the ASP processor with the selected wordlist in the search order. The ASPX file that generated the response above could have been as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<% language=forthscript %>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
>

  <soap:Body>
    <HelloWorldResponse
    xmlns="http://oxns.demon.co.uk:37851/">
      <HelloWorldResult>
        <% /s1 .Param ." :" /s2 .Param ." :" /i1 .param %>
      </HelloWorldResult>
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

The key line is this one:

```
<% /s1 .Param ." :" /s2 .Param ." :" /i1 .param %>
```

This text between **<%** and **%>** is treated as Forth sourced and **EVALUATE**d by the Forth interpreter. Any output from the Forth is simply sent to the socket to which the response text goes.

XML input design

XML is an enhanced HTML with much stricter rules. In particular, every opening tag must have a closing tag. The major difference is that the choice of tag names is up to you, and each section of an XML document forms a tree.

As with HTL, you can choose to ignore tag pairs. We chose to process input the tags at the closing tags because the text was available as the content of the tag.

```
      <s1>string</s1>
      <s2>string</s2>
      <i1>int</i1>
```

Thus, for output we need to be able to access the tags by name, and to extract the data from them. The simplest way is to create Forth words for tags we wish to process, and to ignore all others. In the output phase, we can then use these Forth words.

All Forth words corresponding to tags share a common data structure which controls how data is set and displayed. To ease construction of services, a source notation was devised. An example is:

```
[services
  [service Service1
    Xcstring: /s1
    Xcstring: /s2
    Xint: /i1
    Xint: /i2
    Xfloat: /fl1
    Xoperation: HelloComplexWorld
    Xoperation: HelloLong
    Xoperation: HelloWorld
    Xoperation: HelloInts
    Xoperation: HelloFloat
    Xoperation: HelloDouble
  service]
services]
```

Several services can be available from one server. A service description consists of the data it uses and the operations it supports.

XML parser implementation

We started from Jenny Brien's code published in ForthWrite, the magazine of the UK Forth Interest group. It was later reimplemented by Leo Wong, with extensions for handling attributes, which are the name/value pairs (`name="value"`) found after the tag names inside a tag declaration.

The code has been extensively rewritten to add error checking and to deal with more cases which were discovered when we exported a large Excel spreadsheet to XML.

The intention of the original code was to be able to **`include`** an XML file as a Forth source code file. This makes testing easy, but has limitations when dealing with web services as the HTTP headers have to be bypassed. The solution was to provide a version of **`include`** that we call **`IncludeMem`** `( caddr u -- )` which performs the function of include from a block of memory. This word has other uses in embedded systems and is sufficiently useful that we incorporated it into the VFX Kernel, not least because the word has carnal knowledge of the kernel.

The final code code can be found in the file *Lib\XML.fth* in all VFX Forth for Windows distributions.

XML output design

Output of XML for web services is defined in ters of standard data types. Having defined a structure for each data item, we can insert the correct output routine when we define an instance of a data type. In the ForthScript below

```
<% /s1 .Param ." :" /s2 .Param ." :" /i1 .param %>
```

the words **/s1 /s2** and **/i1** correspond to the closing XML tags `</s1> </s2>` and `</i1>`. The word **.Param** displays the data in XML format.

The only issues here are in matching the specification, and in converting the XML special characters such as the '<' and '>' characters to their XML representations.

Testing against an existing web service

The parser was tested by constructing an example web service using the Microsoft C#.NET toolchain. This showed what we can expect from other systems. We could then test Excel against this web service. By logging the transactions, we could see what was expected.

The objective of our first server was thus to replicate the test server.

Required changes to PowerNet

The major changes to PowerNet were in the detection of a web service request. We handled this by using the ASMX extension for web service files. Another change was that most web service requests are made as POST requests, whereas most web pages are served as GET requests. GET requests are used when the state of the server will not change. Since web services exchange and modify data, the state of the server can change, and so POST requests are used.

We also had to modify the CGI handler to recognise the "?wsdl" string which is discussed below.

WSDL files

It is all very well to be able to exchange data, but you also have to be able to publish **how** you are going to exchange the data. This is handled by the Web Services Description Language (WSDL, or "wizdl"). Every web service includes two files which can be accessed by GET requests.

The first is a standard web page which tells humans how to use the service and often includes software documentation. The second is an XML description of the service as an XML schema.

In the real world

So does it really work? Yes, it does. There were the usual problems with Excel not being totally standards compliant, especially in terms of requiring "keep-alive" connections. However, since these restrictions are the result of recommendations in later versions of the HTPP RFCs, these problems may be forgiven even if they cause problems in low-resource environments such as are found in embedded systems.

Embedded systems

Much as we have complained about the behaviour of Excel, it is in reality the application that most people want to be able to exchange data with. PowerNet v3 can be coerced to work with 16k of RAM in total, is much more comfortable in 32k, and surprising in 64k bytes. The effect

of keep-alive connections with Excel is that you have to generate XML output data (or at least know its size) before generating the HTTP header.. This requires either more code or more RAM. We simply chose the more RAM route for the PC implementation.

Future developments

We intend to port PowerNet v4 to embedded systems, and to enhance the ease of use by automating the generation of the response scripts and WSDL files.