OpenVMS Technical Journal V7

WASD in SOAP/XML Transaction-Oriented Environments

Authors: Mark Daniel, Jeremy Begg, Ben Burke, Howard Taylor

Overview

The WASD VMS Hypertext Services Package is a general-purpose Web service with sufficient additional specialized capabilities to find a niche in transaction-oriented network services. This article explains how to use WASD to deliver SOAP remote procedure calls. The first section describes the persistent scripting mechanism provided by WASD. This is followed by two case studies that explain the development of SOAP-based transaction systems in which WASD provides the Web infrastructure to interface front- and back-end processing. The case studies describe technology choices, implementation details, system performance, and general outcomes.

WASD PERSISTENT SCRIPTING

Author: Mark Daniel

Initial development of the WASD VMS Hypertext Services Package began in 1994. This was an early venture into the then-promising new world of distributed networked information resources. To ensure efficiency and low latency, the key design decisions were:

- Use as few layers as possible between the Web services and VMS
- Use VMS's AST delivery mechanism for servicing multiple, concurrent requests in an event-driven manner
- Provide extensibility by using an effective, native scripting environment

There was no concern about making the solution "too closely integrated with VMS"—it couldn't be too close! Finally, I was free to play with as much of OpenVMS as I possibly could. I particularly liked that one.

After eighteen months of development, the HFRD VMS Hypertext Services Package (as it was then called) was released as open source with the OpenVMS Freeware CD v3.0. At that point, it had much of the core functionality and support programs that is in WASD today, such as concurrent request processing, file serving, CGI scripting environment, Conan the Librarian, and Bookreader. After the product was released to the public, development became user-initiated. In subsequent years, SYSUAF authentication, SSL, and proxy services were introduced, as well as a host of refinements to existing functionality. Thanks to a truck-parts supplier in Michigan, who in 1997 suggested the need

for a low-latency, high-efficiency, persistent scripting environment, the WASD facility now called CGIplus was developed.

Persistent Processes

WASD scripting was originally written using a generic CGI mechanism. CGI scripting is a standards-defined method for providing request data to the scripting environment and for conveying the response to the server and returning it to the client. This method has traditionally required the creation of a new process for each request, resulting in three critical performance issues:

- OpenVMS process creation is notoriously expensive. Dealing with this required the elimination of unnecessary process creation. In practice, most scripts do not alter the process environment significantly enough to warrant disposing of the process itself. CGIplus allows successive CGI scripts to reuse the process, after some minimal housekeeping each time that the server manages the process from idle to occupied and back to idle again.

  There have been a number of proprietary variations on CGI designed to improve performance through process context persistence (for example, ISAPI and FastCGI). CGIplus is similar, but more straight-forward than these approaches. In fact, the WASD package provides implementations of ISAPI and FastCGI built on its own CGIplus technology.

- Many scripts are expensive to instantiate. To resolve this, the script instantiates itself and its required resources within the process and executes multiple requests without performing significant initialization each time. (This is different from executing multiple different scripts through the one process.) Other resources that are used by scripts, such as databases, have their own instantiation and rundown demands. Some scripting environments therefore perform best if a script is started the once and then used repeatedly over multiple requests. Under WASD, the server manages which script is currently in use in which process, and directs relevant requests to that process. This is known as a CGIplus script.
- Interpreters like Java, Perl, PHP, and Python engines require significant resources and exhibit noticeable latency when initializing. To resolve this issue, CGIplus allows the scripting engine to be given successive, different scripts to execute without the requirement to reinstantiate the underlying interpreter each time. Again, the server manages which processes have which engines, and allocates one appropriately to each request. This is called the CGIplus Run Time Environment (RTE).

Use of CGIplus scripts and RTEs provides up to ten times the reduction in script latency (yes, that's 10x!) with the resulting improvement to throughput. The cost of this solution is a slight increase of complexity in coordinating script initiation and in obtaining the CGI variable information using a separate data stream. CGIplus is CGI, plus lower latency, plus greater throughput, plus far less system impact.

CGIplus Operation

The WASD CGI implementation defines a collection of environment variable names containing associated strings, providing a representation of the request parameters. These are known as the CGI variables, which WASD implements under standard CGI using DCL symbol names and values. A set of standard responses convey HTTP status information about the relative success of the request and any associated response content.

All WASD CGI/CGIplus/RTE scripts (with the exception of DECnet/OSU emulation scripts) use mailboxes for Inter-Process Communication (IPC) between server and script. These record-oriented, general-purpose communication devices are suitable for I/O in all OpenVMS environments (DCL, standard utilities, compiled programs) and made available using SYS$OUTPUT, SYS$INPUT, and CGIPLUSIN. WASD automatically adjusts carriage-control for output according to response content type, and also allows you to make fine adjustment of such behaviors under script control. Standard CGI scripts require no modification for use with WASD.

A CGIplus script is indicated to the server by a configuration mapping rule that modifies the management of the script and its associated process. With CGIplus, the request data is provided to the script using the same set of familiar CGI variables. However, instead of being implemented using DCL symbols, the CGI variable names and associated values are written by the server to the CGIplus

process and read as a succession of records from the CGIPLUSIN stream. An initial sentinel record prepares the script for a new request. After that, each record contains a name-value pair until the receipt of an empty record indicates end of request data — a very simple protocol. Based on the supplied request data, the script generates a response using the standard CGI schema, terminating it with an end-of-output sentinel record. The script then enters a wait state until the server provides another initial sentinel record at the beginning of the next script request. This cycle repeats itself until the script itself exits or until a configurable period passes without a fresh request being directed to the script process. When that period expires, the server runs down the script process. The process is simple, elegant, and very efficient!

CGIplus is a straightforward variation on standard CGI scripting that allows DCL implementation. The following example from the WASD package shows all the functional elements of the operations described, and provides a plain-text list of the CGI variables associated with the request, along with the number of times the script has been invoked.

```
$! Simple demonstration that even DCL procedures can be CGIplus scripts!
$! 08-JUN-1997  MGD  initial
$!
$ say = "write sys$output"
$ UsageCount = 0
$ FirstUsed = f$time()
$ open /read CgiPlusIn CGIPLUSIN
$!
$ RequestLoop:
$!
$!   (block waiting for request, this initial read is always discardable)
$    read CgiPlusIn /end=EndRequestLoop Line
$    UsageCount = UsageCount + 1
$!
$    say "Content-Type: text/plain"
$    say ""
$    say "Number of times used: ''UsageCount'"
$    say "First used: ''FirstUsed'"
$    say "Now: ''f$time()'"
$    say ""
$!
$!   (read and display the CGIplus variable stream)
$    CgiVarLoop:
$       read CgiPlusIn /end=EndCgiVarLoop Line
$       if Line .eqs. "" then goto EndCgiVarLoop
$       say Line
$       goto CgiVarLoop
$    EndCgiVarLoop:
$!
$    say f$trnlnm("CGIPLUSEOF")
$    goto RequestLoop
$!
$ EndRequestLoop:
```

A working example is available from http://wasd.vsm.com.au/cgiplus-bin/cgiplusproc. The same elements and structure can be seen in Case Study 1 (the telecommunications industry case study) and in Case Study 2 (implemented in GT.M (or MUMPS)).

Example code in C, Perl, and Java is provided with the WASD package, as well as a C library for transparently handling CGI and CGIplus scripts. A more efficient variant of the CGI variable transfer, struct mode, further improves CGIplus performance by up to another 2x!

CGIplus is the basic construct used by developers to implement persistent scripting environments and keep specific resources instantiated. The server also carefully controls script shutdown. Scripting environments need to be run down for various reasons, such as usage limits, processing errors, application shutdown, or resource exhaustion. The WASD server attempts to allow scripts to elegantly release instantiated resources by using the $FORCEX system service to invoke exit handlers. It subsequently uses $DELPRC to shut down any particularly recalcitrant scripts. This staged approach

permits databases to be released without rollback and to meet similar rundown requirements to be met.

During 2000, the WASD CGI/CGIplus/RTE scripting infrastructure was revised significantly to allow the management of detached and non-server account processes. Using completely detached processes removes issues that are associated with pooled quotas because each process is a completely independent scripting entity. Detached processes also allow scripts to be run in their native account environment, if necessary. These facilities are available on all platforms and versions of OpenVMS from V6.0 through V8.2-1.

CASE STUDY 1: TELECOMMUNICATIONS INDUSTRY

Authors: Jeremy Begg and Ben Burke

This case study describes a WASD+SOAP installation at a large Australian telecommunications carrier that provides landline and cell phone services and Internet services. Due to Commercial in Confidence considerations, we cannot identify this customer.

Site Overview

The billing system for Postpaid Mobile business has been in continuous use for over a decade, providing integrated rating, billing, and online provisioning to network elements that make up the GSM network. Although originally built as an OpenVMS/VAX-only configuration, the billing system has been expanded to include other platforms, including OpenVMS/Alpha. The VAXes are now primarily being used to interface to the GSM network hardware. Recent changes and enhancements to the company's internal systems required a new data schema to encapsulate the entire profile of a customer's service and a way to distribute the data between the different operating systems and applications in use. This schema, the Service Profile, includes billing product information, discounts, promotions, and mobile features information.

To permit existing corporate systems and middleware to exchange this Service Profile information, we decided to use web services technology based on XML, SOAP 1.1, and HTTP. The web services operations for exchanging XML Service Profile information to and from OpenVMS was implemented in just four APIs:

- SetServiceProfile, a SOAP service for processing changes to the database & GSM network
- GetServiceProfile, an HTML form for retrieving a current Service Profile document for any mobile service
- GetProvisioningXML, a SOAP service by which certain non-OpenVMS systems retrieve information to perform network provisioning
- SetCompletionStatus, a SOAP service used by those systems to inform the billing system that the network provisioning has been completed (successfully or otherwise)

The APIs had to run on both OpenVMS/Alpha and OpenVMS/VAX. However, HP's Secure Web Server (based on Apache) was not available for VAX. Furthermore, all of HP's standard offerings in the web services area were based on Java – also not available on VAX. Therefore we built our own solution using available technology.

Selecting the Technology

First we needed a web server. WASD is a widely used web server available on both OpenVMS/VAX and OpenVMS/Alpha. The following features made it a good choice for us:

- The CGIplus architecture permits us to satisfy API requests without the overhead of per-request process creation. This was very important to us because these APIs are expected to process tens of thousands of transactions per day.
- The WASD implementation includes monitoring and troubleshooting capabilities. Because we standardized on a single Service Profile document schema, our APIs were complex. The WASD facilities enables us to debug during development and to monitor during implementation.
- WASD includes support for most commonly used web technologies (for example, SSL).

Next, we needed an XML parser. The Gnome libxml2 toolkit was an open source XML parser and library written in ANSI C and available for both OpenVMS/VAX and OpenVMS/Alpha. We found that libxml2 is both rich in features and very efficient. It was not necessary evaluate any other XML parsers.

Writing a SOAP Processor

The SetServiceProfile API is the most complex of the four APIs. This API is implemented as a SOAP 1.1 POST request. SetServiceProfile reads a SOAP document, then calls libxml2 routines to parse the SOAP envelope header and to extract the ServiceProfile body. The header of the request contains various mandatory items such as:

- The phone number(s) being operated upon (there can be up to three—for speech, fax, and data)
- The transaction ID
- The date on which the change should take effect.

The body of the request contains the ServiceProfile document itself: a description of the service offerings that are enabled and disabled.

Textbooks on XML and SOAP suggest that, to process XML documents packaged inside SOAP envelopes, you need a fully web services-compliant environment with the ability to read Web Service Description Language (WSDL) documents, determine on-the-fly what a particular piece of XML is describing, and implement the business logic. However, we were able to implement SOAP processing simply by using the WASD web server and the libxml2 library.

Each of the APIs is coded following the standard CGIplus program structure:

```
Initialize
Loop
   Wait for request
   Process request
   Send response
Until (time to exit)
```

In our application, the "Process request" portion included ORACLE Rdb database activity that uses existing routines written in COBOL and BASIC. We coded the CGIplus programs in C, from which we make the calls to the COBOL and BASIC routines.

The other APIs were more simple than SetServiceProfile, and were therefore implemented using HTML constructs with parameters passed using GET. For example, GetServiceProfile accepts a phone number and historical date, looks up the database to determine the services subscribed to that phone on the requested date, and uses the libxml2 routines to assemble the Service Profile in XML format. The completed Service Profile is then returned to the requesting system.

WASD Features

WASD has some unique features that greatly facilitated both development and implementation, including process termination, proxy authentication, and script monitoring.

Process Termination

The script process does not terminate between requests; therefore, overall throughput and response times is very good. WASD provides mechanisms for gracefully shutting down inactive CGIplus processes. However, we implemented the following mechanisms for shutting down the program proactively:

- Limit the number of iterations of the main processing loop. This mitigates the possible effects of memory leaks, unreleased database locks, and other latent programming errors in the legacy database routines.
- Exit the program if an error is detected in the input XML or during database operations. This minimizes the disruption to service when a error detected in one request causes a later request to fail without warning.

During testing, we discovered a second form of process termination for which we have no real solution. When WASD creates a process to run the CGIplus program, SYS$INPUT and SYS$OUTPUT

are mapped to VMS mailboxes that are used by WASD to communicate with the process. WASD discards all process output received before the first request is accepted by the process, which avoids the problem of miscellaneous output from SYS$SYLOGIN and LOGIN.COM. However, after the process has accepted a CGI request, it conforms strictly with CGI scripting protocols. The next output by the process after it has read the request parameters must be a CGI or HTTP response header. If it is not, WASD runs down the script and terminates the process. As a result, debugging is difficult because only the first line of output is recorded in the WASD process log file.

Proxy Authentication

The billing applications maintain audit trails that include the name of the OpenVMS user who changes the database. HTTP requests are performed on behalf of mobile phone resellers, and each reseller is mapped to a specific OpenVMS user account. Therefore, the CGIplus process calling application specifies the identity of the reseller in the request. (In this context, the term "reseller" refers to a chain of stores operating under a single brand, not an individual retail outlet.) To ensure that the CGIplus process was created in the appropriate OpenVMS account, authenticated HTTP is used to make the request, forcing the client to include an authorization header in the HTTP request. The WASD configuration runs the script OpenVMS user name specified in the HTTP authorization header.

WASD's Proxy Authentication mechanism maps HTTP-supplied credentials to OpenVMS user names to avoid having to maintain valid OpenVMS user name/passwords combinations on the remote system. Proxy Authentication uses the following configuration files:

- A plain text authentication file, which lists the remote user names and their corresponding passwords
- A proxy mapping file, which associates the HTTP-supplied user names to one or more OpenVMS user names

If the remote system's HTTP request supplies a valid username and password from the authentication file, WASD Proxy Authentication maps the request to the appropriate OpenVMS user account.

Script Monitoring

WATCH is an excellent WASD facility that shows the processing of requests in detail. The incoming request header and body are optionally viewable. The generated response (header and/or body) can be viewed. The associations and authorization rules, CGI variables, and so on, are available at run time and can be viewed using a web browser interface. WATCH makes it much simpler to debug web scripts.

WASD meets the requirements of this environment with quality monitoring tools, features, documentation, and support. It is unlikely that any other webserver, on any platform, would be as good a match in all these areas.

CASE STUDY 2: COAST CAPITAL SAVINGS

Author: Howard Taylor

Coast Capital Savings is a credit union servicing 300 000 customers in the Lower Mainland and southern Vancouver Island regions of British Columbia, Canada. Coast Capital Savings' banking system runs on OpenVMS AlphaServers and is written in Greystone Technology M (M). The banking system is natively accessed through a VT-terminal interface. In 2002 we (the Information Technology Group of Coast Capital Savings) were asked to deliver a GUI interface to the banking system. After evaluating various proposals, we decided to build a Windows .NET client application and interface.

From SOAP to M

To make it easier to reuse existing banking system routines and to ensure that the banking application's business rules were enforced, we chose to implement a remote procedure call (RPC) mechanism. Of the available RPC mechanisms, including CORBA, Microsoft DCOM, and JavaBeans, the one which seemed to be the most flexible for a cross-architecture scheme such as ours was the XML-based SOAP-RPC.

We needed to take an XML SOAP-RPC request and transform it into a data structure that could be read by GT.M. To illustrate, here is an example of a SOAP-RPC procedure "GetTranCodes" with a single parameter "CustomerID" equal to "123456789":

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
 <env:Body>
  <p:GetTranCodes xmlns:p="http://coastcapital.local/ibsclient">
  <p:CustomerID>123456789</p:CustomerID>
  </p:GetTranCodes>
 </env:Body>
</env:Envelope>
```

This XML structure does not map directly to an M structure, so we mapped the procedure name and parameters into an M local array as:

```
LOCAL("ProcedureName")="GetTranCodes"
LOCAL("Body","GetTranCodes","CustomerID")=123456789
```

The XML namespaces and their prefixes wer mapped as:

```
LOCAL("Envelope","~xmlns","href")="http://www.w3.org/2002/06/soap-envelope"
LOCAL("Envelope","~xmlns","prefix")="env"
LOCAL("Body","GetTranCodes","~xmlns","href")="http://coastcapital.local/ibsclient
"
LOCAL("Body","GetTranCodes","~xmlns","prefix")="p"
```

We had already installed the Gnome libxml2 parser on our OpenVMS systems, so we built a C external routine to use the libxml2 API to parse SOAP-RPC XML documents into M local variables as shown above. M code then:

- Reads the local array.
- Accesses a table in the banking database to validate the RPC name and arguments.
- Invokes an M routine referenced by the RPC procedure name.
- Formats the results or errors into the response format specified in the SOAP 1.2 recommendation.

Integrating with WASD

We needed a robust, high-performance delivery mechanism for this new application. We were already using the WASD web server on our system for serving up BookReader documentation, so we investigated using WASD to deliver SOAP-RPC web services.

We were pleasantly surprised by the rich functionality we found, including:

- CGI scripts can be started from a DCL command procedure
- CGIplus avoids the penalty of OpenVMS image activation
- WASD can run detached CGIplus processes under different user accounts
- CGIplus processes can be gracefully shut down to permit dynamic software updates
- Load-balancing and throttling are built in

All this functionality came with excellent performance. A .NET single-stream test application we created to randomly exercise 47 of our commonly-used RPCs reported an average round-trip response of 128 ms (minimum 16 ms).

Programming WASD's CGIplus interface is very straightforward. This small amount of M code performs all of the interfacing our RPC server needs to do with WASD:

```
        S CGIIN="CGIPLUSIN"                ; CGIplus standard input
```

```
        S CGIOUT="SYS$OUTPUT:"             ; CGIplus standard output
        O CGIOUT:(BLOCKSIZE=65535)         ; Open input
        O CGIIN:(READONLY:BLOCKSIZE=65535) ; Open output
        F  U CGIIN R LINE Q:$L(LINE)=0  D  ; Loop reading input until blank line
        . S KNAM=$P(LINE,"=",1)            ; Get key name of CGI variable
        . S KNAM=$E(KNAM,5,$L(KNAM))       ; remove "WWW_" prefix
        . S WWW(KNAM)=$P(LINE,"=",2)       ; Store value in local array WWW
EOF     ;
        I WWW("REQUEST_METHOD")="POST" D POSTIN ; If POST input, do POSTIN
        U CGIOUT W $ZTRNLNM("CGIPLUSEOF")  ; Send EOF (required by CGIplus)
        K (CGIOUT,CGIIN,NULLDEV)           ; Clean up
        Q
    ;
POSTIN  ; Read POST input
        ;
        S HTTPIN="HTTP$INPUT"               ; POST input stream
        O HTTPIN:(READONLY:BLOCKSIZE=32767) ; Open POST
        S POST=""
        U HTTPIN:EXCEPTION="G EXIT"
        ; Read contents of POST into a local variable
        F  Q:($L(POST)=WWW("CONTENT_LENGTH"))  R REC S POST=POST_REC
        C HTTPIN                            ; Close POST
        . . . (process the POST contents)
        Q
```

Using a .NET class to interface to the banking system's SOAP-RPC server, we were able to quickly build our Windows GUI client application in VB.NET. We were also able to reuse the .NET class in other client-server applications that needed to communicate with our banking system.

The WASD SOAP-RPC mechanism implemented at Coast Capital Savings today serves approximately 1500 interactive workstations, as well as a busy customer-facing IVR system, and many new applications are scheduled to use it in the near future. Overall we have been impressed by the ease of implementation, high performance, and minimal overhead of this excellent web server.

For more information

The WASD package is demonstrated and available at http://wasd.vsm.com.au/.

The libxml2 toolkit is described and available at http://www.xmlsoft.org/.

SOAP and its RPC are described at http://www.w3.org/TR/soap12-part2/.

Jeremy Begg may be contacted via the VSM site at http://www.vsm.com.au/.