OpenVMS Technical Journal

Examining Web Services:

Protecting Your OpenVMS Investment

David J. Sullivan, Expert Member Technical Staff

What are web services?

Web services are a set of technologies designed to aid in the development of heterogeneous and platform-neutral solutions. Since their introduction, web services have received unprecedented acceptance among enterprise software vendors. They provide businesses with a strategic advantage because of their relative simplicity, lower cost, and wide cross-platform availability.

This article examines web services from both a business perspective and a technical perspective. It explains the business needs driving web services as well as the core web services technologies. It also explains the OpenVMS strategy for supporting web services and provides example applications that illustrate the integration of an OpenVMS application with other platforms, such as Microsoft® .NET and Linux®. These topics may be read independently. Use the links below to go to the sections of most interest to you:

Why should you care about web services?

Traditional integration technologies have many flaws. The traditional approach to integration relies on the use of middleware products. These products are often costly, proprietary, and difficult to use. Middleware is rarely available across all platforms and the costs associated with the use of middleware are significant. Perhaps of most concern is that a user is forced to bet their business on the success of a particular vendor's middleware product.

Web services provide a technology that is capable of tying together any applications written in any language on any operating system. As compared with traditional middleware, web services are less expensive, open, and simpler to use. Your business is safe because implementations of the web service standards are widely available from many different sources.

A common and costly business problem occurs when technology limits the availability of information that flows throughout the corporation. Often, applications located in different divisions of a corporation cannot share data because they use different operating systems, middleware, and programming languages. These islands of technology are often isolated from each other or use gateways that attempt to patch together the various technologies. The potential combinations of integration points are significant.

For example, in Figure 1, the lettered boxes represent separate applications, and each arrow represents a technology used to integrate them. The boxes represent the following types of applications:

A: OpenVMS applications

B: Microsoft C++ application using DCOM

C: IBM C application using MQSeries

D: HP-UX COBOL application using CORBA

E: This system does not access an OpenVMS system

F: Microsoft Windows® Java™ application using JMS

G: Sun Java application using J2EE

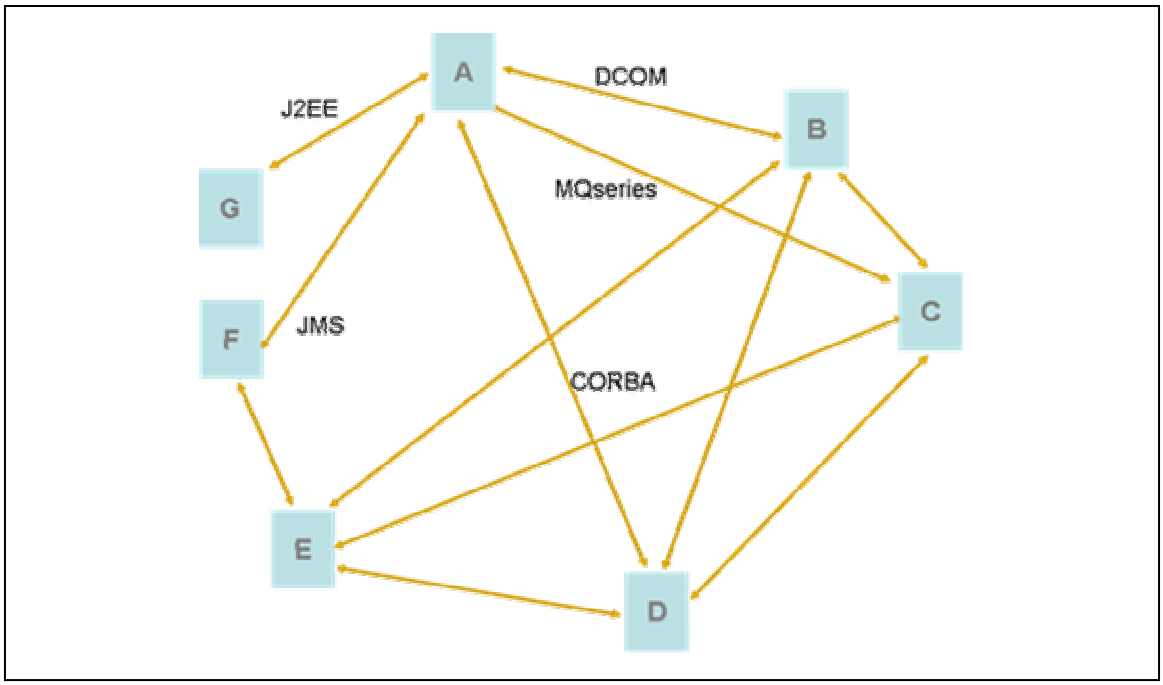

Figure 1: High-cost, complex IT environment

It is easy to see the complexity and overhead that is required to develop and maintain this infrastructure. Each arrow represents a large investment of resources in hardware, software, management and training. Each hard-coded connection has unique security concerns that increase complexity. Also, this outdated technology does not work well across the Internet, thereby eliminating the ability to perform effective business-to-business interactions. Integrating applications becomes very difficult because each pair of applications must be addressed separately.

With web services, solutions architects have a common integration technology on all platforms. By exposing an OpenVMS application as a web service, it can be called by any web service client, regardless of the platform or programming language used by the client (Figure 2). Likewise, the web service client never knows the platform or programming language of the web service being called.

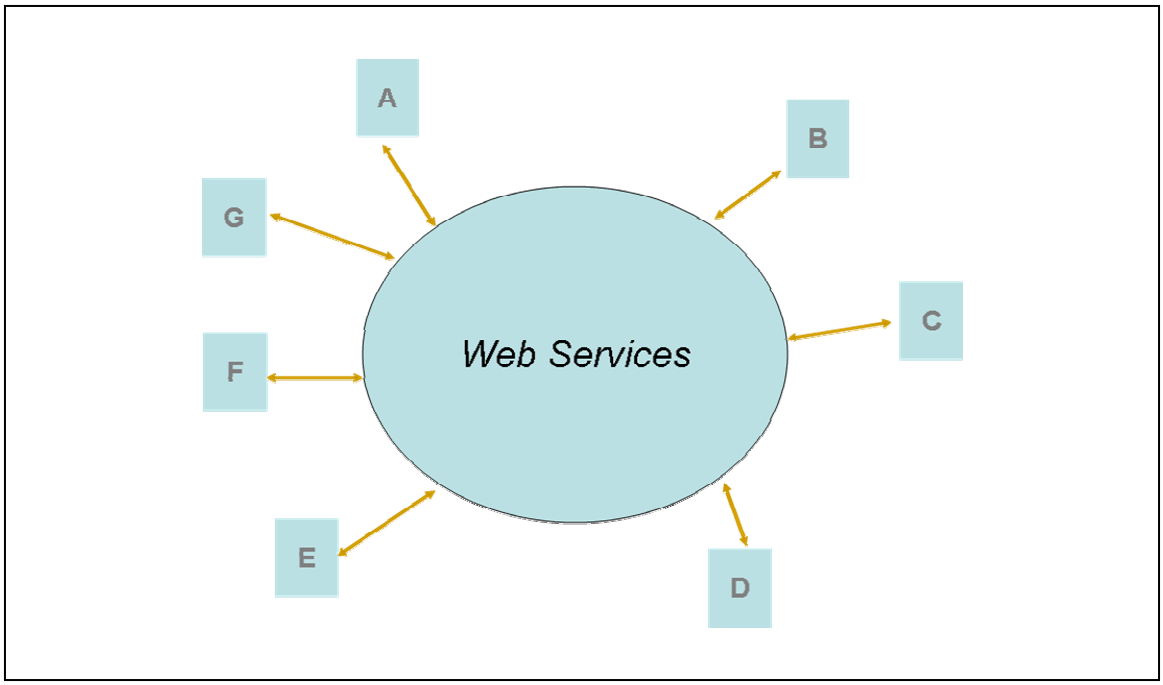

Figure 2: Simpler, less expensive IT environment

Integration issues with OpenVMS are now much easier and less costly with web services. Architects can make use of the unique strengths of OpenVMS systems without having to make the undesirable tradeoffs listed earlier. Legacy applications can be integrated with other solutions that are based on web services. The original investment in design, development, testing, and maintenance of OpenVMS applications continues to pay dividends to the business while freeing money and resources for investments in other areas.

Table 1 lists the advantages of using web services over traditional middleware integration products.

Table 1: Advantages of web services over traditional middleware

Traditional middleware	Web services
Communication styles are dictated by middleware. Usually RPC or MOM.	Supports many styles of communication RPC, Messaging, 1-way, 2-way, conversational.
Lack of cross-platform availability creates islands of separate technology.	Allows single integration backbone across all platforms.
Lacks standards: • Many ways to do common jobs • Proprietary solutions from individual vendors	Standards based: • One way to do common jobs • Open solutions from many vendors
Hard to use (design, program, manage).	Simpler to use.
Expensive.	Less expensive.
Does not encourage reuse of code.	Encourages reuse of code.
Customized for certain operating systems and programming languages.	Operating system and programming language neutral.
Does not scale across devices.	Can be used from the smallest handheld device up to a large server.

Does not work over Internet.	Designed to operate over Internet.
Brittle and rigid.	Flexible and adaptable.
Tightly coupled.	Loosely coupled.
Proprietary data formats.	Leverages XML.
Undocumented binary protocols.	Uses open and standard text protocols.
Invasive.	Less invasive.

How do OpenVMS customers benefit?

There are many ways in which OpenVMS customers can benefit from using web services. The advantages over traditional middleware are significant.

Investment protection

An existing application can be exposed as one or more web services to extend its use and prolong its life.

Utilization of OpenVMS strengths

The strengths of OpenVMS can be used throughout a corporation to host critical, bet-your-business applications. These OpenVMS applications can be exposed as web services and can be used by non-OpenVMS platforms. The client code never knows that the service being used is deployed on OpenVMS.

Availability of more applications

OpenVMS applications can make use of applications that are not natively available on OpenVMS. In these cases, the OpenVMS system acts as a consumer of web services that are deployed on other platforms, such as Microsoft Windows .NET or HP-UX.

Phasing out expensive middleware

Customers have the option of phasing out costly, unnecessary middleware over time.

New opportunities for legacy applications

Web services provide a new opportunity for applications that previously could not be integrated with traditional middleware. Web services support more design models than traditional middleware and are neutral as to the programming model used by the application.

Cross-language programming

Web services can be used on OpenVMS systems as a simpler and more extensible mechanism for communication between different programming languages.

Availability over the Internet

OpenVMS applications can be programmatically accessed from the Internet. Web services are designed to operate over the Internet without any extra design or coding.

Cost savings

Standardizing on web services reduces the cost of development, testing, maintenance, and management. Web services platforms are widely available from many sources including free, high-quality, open source offerings.

Compatibility with newer technologies

Because of the wide acceptance and availability of web services, new technologies such as the Adaptive Enterprise, Virtualization, and the GRID are all built on a web services IT backbone.

Use of XML

XML is the preferred data storage and description mechanism. It is highly likely that a business already uses XML documents to represent valuable data. Web services are based on XML and naturally support the exchange of XML documents.

Common business languages

Industries are using XML to define industry-standard definitions of common business entities.

Success Stories

Many interesting case studies show how web services have had a significant and positive impact for Fortune 500 businesses. *Web Services Case Studies*

Technology

The technologies behind web services are designed by standards organizations. There are a staggering number of drafts, recommendations, and standards dealing with web services. Fortunately, most developers do not need to master, or even understand, the majority of these specifications.

Web services development platforms hide many of the details of these technologies from the developer. In fact, some development platforms allow engineers to use wizards to generate all the source code for a web service. However, the more developers understand the concepts behind the core technologies, the more effectively they can design and develop robust solutions based on web services.

Each web service platform presents these same concepts in different ways and with different levels of exposure. After reading this section, you should have a much better understanding of how the unique development tools are able to generate highly interoperable applications.

WS-I

The Web Services Interoperability (WS-I) organization is responsible for delivering clear and consistent recommendations for ensuring interoperability between web services. This is perhaps the most important standards organization in the web services world. In order for web services to continue to spread, there must be a single definition of web services compliance.

In August 2003, WS-I announced the approval of the Basic Profile 1.0 (BP 1.0). BP 1.0 consists of implementation guidelines for how a set of core web services specifications should be used together to develop interoperable web services.

This article describes only those features that are included in the Basic Profile.

XML

XML stands for Extensible Markup Language. It was adopted by the W3C in 1998 and has since become the preferred way to store business data.

XML was designed to describe and store data. It is similar to a markup language (such as HTML) in that it has begin and end tags. Unlike HTML, it has no predefined tags. Rather, with XML the author first defines the tags and then uses them to describe the data. When XML tags are defined, they often have a hierarchy to convey a relationship, such as parent-child.

XML does not determine how the data is displayed. Separating the data from how it is displayed allows XML to remain simple and flexible. XML gets its power and popularity from its simplicity. An XML document is:

- A plain text file
- Human readable
- Understood by all platforms and programming languages

Because all operating systems understand text files, XML documents can be created and modified using a simple text editor. It is this common understanding of text files that allows XML to be used as a common data format across all platforms and languages.

As an example, the XML file (called employee.xml) in Figure 3 should be easy to understand even if you have never before seen XML.

```
<?xml version="1.0"  encoding=”UTF-8”?>
<employee>
      <name>
        <first>David</first>
        <last>Sullivan</last>
      </name>
      <email>davidjsullivan@hp.com</email>
</employee>
```

Figure 3: employee.xml

The employee.xml file in Figure 3 contains five tags: `<employee>`, `<name>`, `<first>`, `<last>`, and `<email>`. Three pieces of data are stored in the file: `<David>`, `<Sullivan>`, and `<davidjsullivan@hp.com>`. This is a complete, well-formed XML file. It has a hierarchy of tags describing that an employee has a name and an email address. Also, the `<name>` tag contains a first and last name. It's that simple!

It would be simple for an application to parse this XML file and do something interesting with the data. For instance, a browser might display the data as part of an employee list, or an application might parse the file to send out automated email notifications.

XML schema

The XML document in Figure 3 is well formed. A well-formed document is syntactically correct. For example, all begin and end tags are properly nested. This XML document is useful in its current state. However, the document is not considered valid because we have no way of knowing whether the tags and associated data are semantically correct. For instance, if we remove the line `<first>David</first>`, is the `<name>` tag still useful (valid)? The answer is maybe. It depends on the intent of the person who defined the tags. This is where an XML schema comes in.

An XML schema is a language used to describe XML tags. When an XML document has a schema, it becomes much easier for an application to understand the layout of an XML document. Also, a schema allows an application to determine whether the contents of the XML file are semantically valid before processing the document.

Each tag in an XML document is defined with an element declaration. When writing a schema file, the author has the option of specifying different properties for the elements, including:

- An associated data type (for instance, string, number, or user-defined type).
- Constrain the valid values for a data type (for instance, allow only integers that are even).
- Require or prohibit child elements and attributes (for instance, the element name could require a last name but make the first name optional).

There are many more options in XML schemas. Refer to a good book on XML for details.

Figure 4 shows the schema file (called employee.xsd) for the XML document in the Figure 3. At first sight, the schema language appears complex (and it is), but after a day or two, writing schema files becomes second nature.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="x-schema/employeeData"
            xmlns="x-schema/employeeData"
            elementFormDefault="qualified">

<xsd:element name="employee">
    <xsd:complexType>
        <xsd:sequence>

            <xsd:element name="name">
                <xsd:complexType>
                   <xsd:sequence>
                        <xsd:element name="first"/>
                        </xsd:sequence>
                        <xsd:element name="last">
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>

            <xsd:element name="email">
            </xsd:element>

        </xsd:sequence>
   </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Figure 4: employee.xsd

In the employee.xsd schema file, we see the definitions of the elements *employee*, *name*, *first*, *last*, and *email*. Notice the nesting of the elements. Any XML file using this schema must include all of the elements defined in the schema, because the author didn't mark any of them as optional. For this reason, the answer to the question in the previous section is no, the <name> tag is only valid if it includes both a first and last name.

Modern development environments, such as the OpenVMS IDE NetBeans, provide rich tools to create and modify XML and XML schemas.

Note: A DTD is an alternative to a schema. DTDs are not addressed here because they are out of date and not used in web services.

Introducing SOAP, WSDL, and UDDI

There are three standards-based technologies that, along with XML, comprise the core of web services.

- SOAP (Simple Object Access Protocol)

  The function of SOAP is to transmit XML messages between two applications. SOAP is itself an XML language and is defined using an XML schema.

- WSDL (Web Services Description Language)

  WSDL is a language based on XML. It is used to precisely define the details of a web

service. Web service consumers use this information to build SOAP messages for the service the WSDL describes.

- UDDI (Universal Description, Discover, and Integration)

  UDDI is a specification that defines a directory for web services. Often called the "Yellow Pages for Web Services," UDDI directories are used to store services that are available to consumers.

Figure 5 provides a high-level overview of how the SOAP, WSDL, and UDDI technologies interact. These three technologies were defined specifically for use with web services. In Figure 5, a web services client performs the following steps to obtain a stock quote from an OpenVMS application that has been exposed as a web service.

Step 1: The client looks in a UDDI directory to find a web service that can supply stock quotes. It discovers the stockQuote service can supply the desired features. The UDDI registry provides an address, in the form of a URL, to the client. (You can skip this step if the client already knows the address of the web service.)

Step 2: The client sends a lookup request to the address obtained in step1, asking for a description of the service. This description is returned to the client in an XML schema language called WSDL.

Step 3: The consumer uses the WSDL description obtained in step 2 to identify the details of the service's functions and associated arguments. The client makes a call to the stockQuote service by sending a SOAP protocol request.

Step 4: The web services platform receives the SOAP protocol request and returns the quote to the client via a SOAP protocol response.

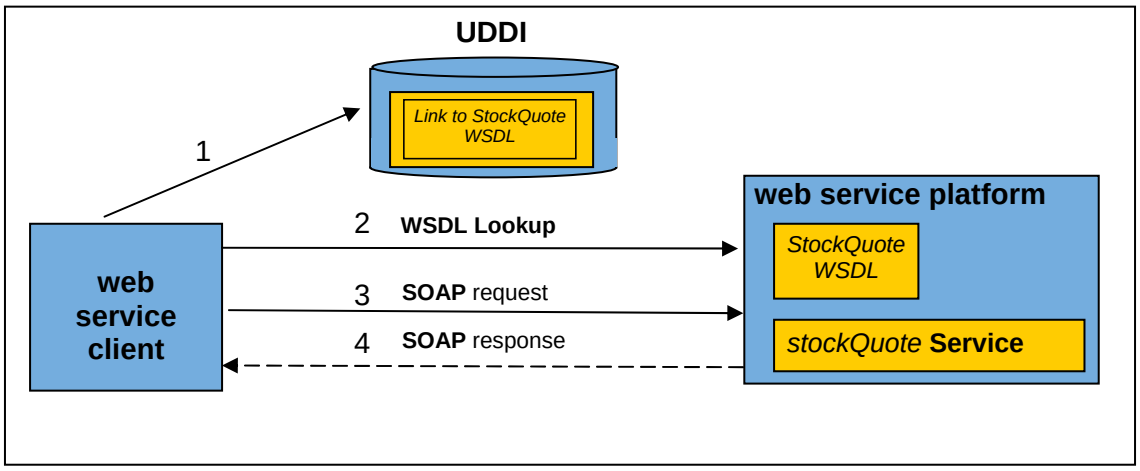

Figure 5: Overview of WSDL, SOAP, and UDDI

SOAP

SOAP stands for Simple Object Access Protocol. The function of SOAP is to transmit XML messages between two applications. SOAP is itself an XML language and is defined using an XML schema. The schema defines a SOAP root element called Envelope. You can look at the SOAP V1.1 XML schema definition at the following URL:

http://schemas.xmlsoap.org/soap/envelope/

Before SOAP can transmit XML to a destination, it must first wrap the XML in a SOAP Envelope. An Envelope defines SOAP-specific elements and has a specific structure. The envelope contains two sections: a Header (optional) and a Body (required). Figure 6 shows the structure of the SOAP Envelope.

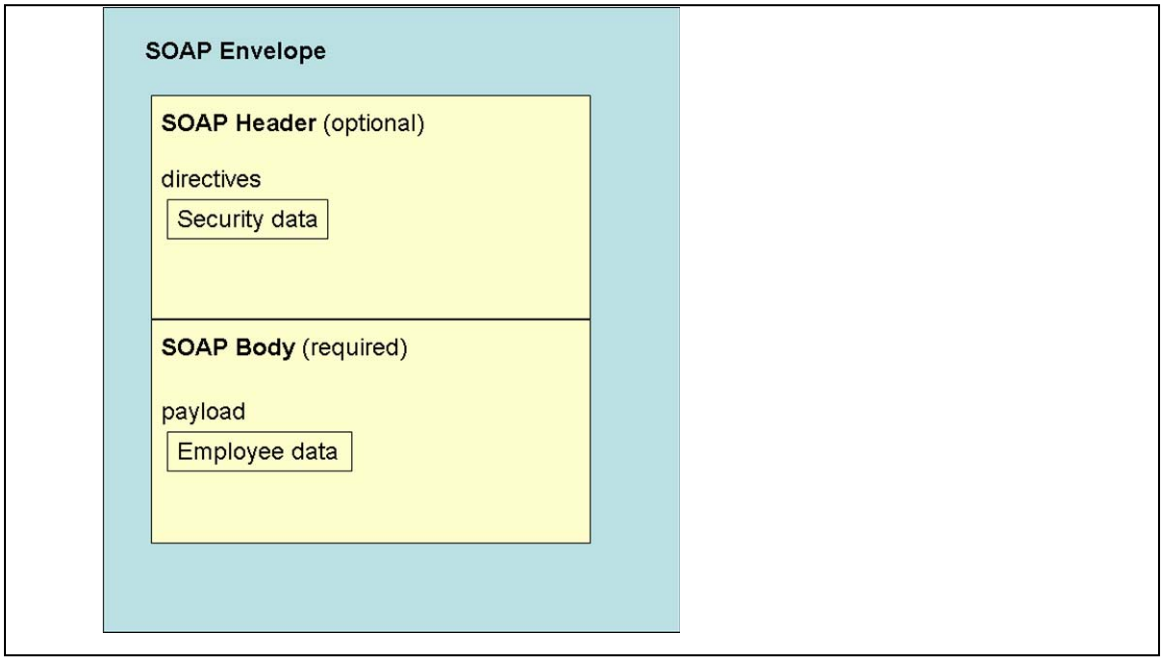

Figure 6: Structure of the SOAP Envelope

Web services platforms use SOAP as the protocol for transmission of XML messages. In Figure 6, the XML employee data is sent within the SOAP Body.

SOAP Messaging Styles

The data of the SOAP Body, also called the payload, is defined by the application. The SOAP schema defines two styles for structuring the payload: RPC and document.

With the RPC messaging style, the structure of the payload is defined by SOAP. SOAP defines its own representation of an RPC call. This structure specifies the method name and arguments for a call to the service.

With the document messaging style, the structure of the payload is defined by the application's XML. SOAP does not impose any structure on the contents of the SOAP Body.

Figure 7 illustrates the two SOAP messaging styles.

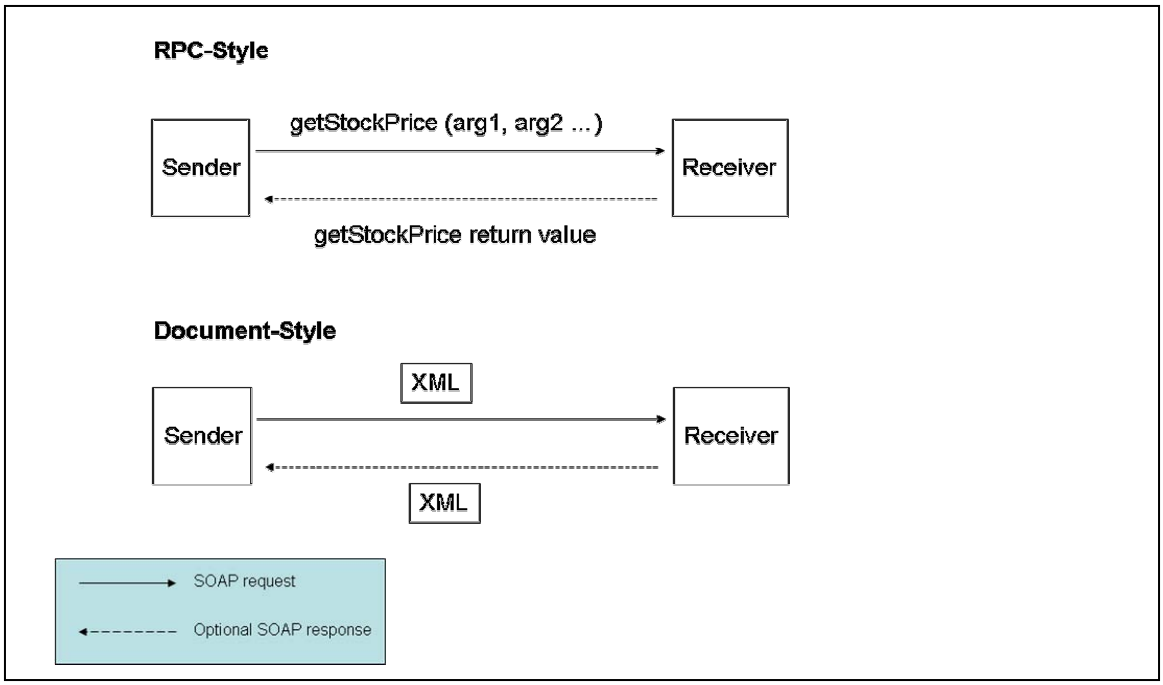

Figure 7: SOAP messaging styles

The following example shows a SOAP request using the RPC messaging style. In this example, the method getStockPrice is being called and requires one argument, which is a ticker symbol. The web services client application does not specify the element names or hierarchy. SOAP automatically uses this structure within the SOAP Body for RPC messaging.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
xmlns:abc="http://abc.com/stock">
<soap:Body>
   <abc:getStockPrice>
      <symbol>HPQ</symbol>
   </abc:getStockPrice>
</soap:Body>
</soap:Envelope>
```

The following example shows a SOAP request using the document messaging style. In this example, the contents of the SOAP Body are defined by the application. The employee XML is sent to the web service without any intervention from SOAP.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
xmlns:abc="http://abc.com/employee">
<soap:Body>
<employee>
      <name>
        <first>David</first>
        <last>Sullivan</last>
      </name>
      <email>davidjsullivan@hp.com</email>
</employee>
</soap:Body>
</soap:Envelope>
```

SOAP Transport

Although SOAP is responsible for transmitting the XML message, it is not designed to use its own network transport to send messages. Instead, SOAP uses the HTTP protocol for sending messages. By using HTTP as the transport vehicle, web services are available across the Internet.

Note: SOAP can use other transports, but the WS-I Basic Profile supports only HTTP.

SOAP extensibility

SOAP is extensible. The header section of the SOAP Envelope is used to add more features. For instance, security, reliability, and transactions are important features for enterprise-level corporations. These features can be added by specifying XML headers in the SOAP Envelope request. The headers can be either standard or propriety extensions to the SOAP protocol.

When a SOAP message is sent to a receiver, the message can be transmitted through one or more intermediaries. An intermediary can examine, modify, and remove headers, depending on its role in the transmission of the message. Figure 8 shows the interaction of SOAP intermediaries.

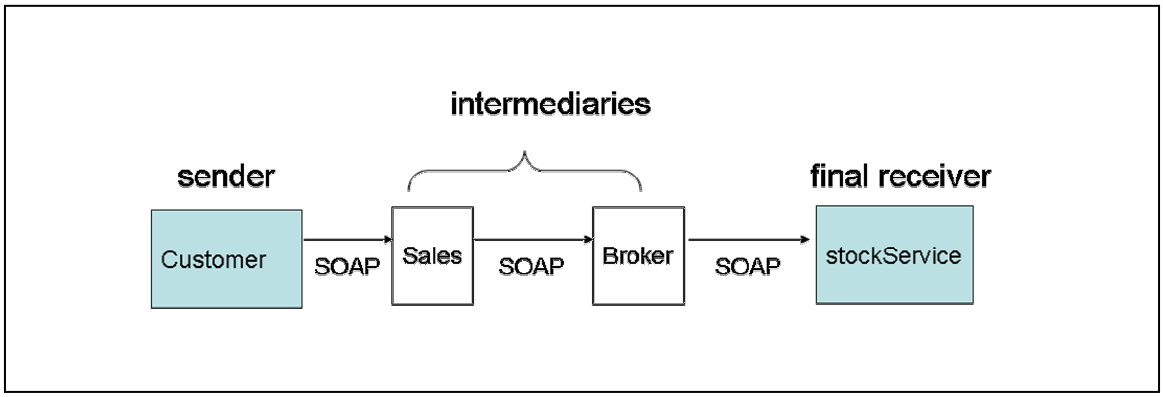

Figure 8: SOAP intermediaries

WSDL

WSDL stands for Web Services Description Language. WSDL is an XML language that is defined using a schema and is used to define precisely the details of a web service. Developers typically do not need to write in WSDL, but they need to understand the organization and purpose of WSDL documents.

WSDL elements

The WSDL schema language provides a number of elements to describe a web service. These elements fall into three groups.

Group 1: These elements describe the interface exposed for use by clients. The interface is defined in abstract terms.

Group 2: These elements "bind" the abstract interface to a transport chosen by the service, such as SOAP. This binding maps the abstract interface descriptions to concrete mechanisms of the chosen transport.

Group 3: These elements name the web service and assign an address that the client uses to identify the location of the service.

Group 1: Describing the interface

The group 1 WSDL elements describe the web service in abstract terms. Table 2 describes these elements.

Table 2: WSDL elements that describe the web service interface (group 1)

Element name	Description
portType	Provides a name for an interface.
operation	Provides a name for a method in a *portType*.
input, output	Describe the *operation* element and are used to specify one of the following interaction models: • 1-way interaction: The client calls the service and the service does not send a response back to the client. In this case. only the *input* element appears in the operation. • Request/reply interaction: The client calls the service and the service sends a response back to the client. In this case, only the *input* and *output* elements appear in the operation.
part	Specifies a piece of data that must appear in the message element. There may be zero or more part elements in a *message* element. There are two attributes for describing the data: **type** and **element**. With the **type** attribute, the *part* element is given an associated data type. The type information is provided to the receiver of the message. With the **element** attribute, the *part* element is an XML document. The document's schema provides the type information for use by the receiver.

WSDL concepts often take some time to comprehend. Let's look at the same concepts from the client's point of view. Imagine that you are a client who wants to call a web service. You might you want the WSDL description to answer the following questions:

- What is the name of the interface for the client to call?
- Within that interface, what is the name of the method to call?
- For the method, what data must the client send, if any?
- Does the service return a response?
- If there is a response, what data will the response contain, if any?

Table 3 describes the WSDL elements associated with this information.

Table 3: Information associated with WSDL elements

Question	WSDL elements	Comments
What is the name of the interface?	*portType*	A servicecan have multiple interfaces, each with a unique name.
What is the name of the method?	*operation*	Each method in an interface has a unique name.
What data must the client send?	*input*, *message*; *part* (optional)	By definition, there is an incoming message; therefore, we need *input* and *message*. The service might not require data from the client, so the message element might not have any *part* elements.
Does the service return a response?	*output* (optional)	Some services don't return a message to the client.
What data will the response contain?	*message*; *part* (optional)	The part dictates what will be returned.

Let's look at fragments of a WSDL document from the ABC stock exchange (Figure 9). They have a web service with an interface named StockQuote and a method named getStockPrice. The method receives an input message that contains a piece of data named symbol, which is a string. The method also returns a message that contains a piece of data named price, which is a float.

```
<!-- message elements describe the data passed as input and output -->
<message name="getStockPriceRequest">
    <part name="symbol" type="xsd:string"> </part>
</message>

<message name="getStockPriceResponse">
    <part name="price" type="xsd:float"> </part>
</message>

<!-- portType and operation element describe the interface -->
<portType name="StockQuote">
  <operation name="getStockprice">
   <input name="symbol" message="abc:getStockPriceRequest"> </input>
   <output name="price" message="abc:getStockPriceResponse"></output>
  </operation>
</portType>
```

Figure 9: Sample WSDL document

Group 2: Binding the interface to a transport

The abstract interface described by the WSDL elements in group 1 must be bound to a transport. The WSDL binding element uses the schema of the specified transport to define a mapping between the abstract interface and the specific transport.

WSDL defines three separate transport bindings, SOAP, HTTP, and MIME. Each transport has its own schema.

Note: This article addresses only the SOAP binding because it is the only binding supported by the WS-I Basic Profile.

The SOAP protocol has a schema for binding to a WSDL interface. The schema defines elements that are used within the WSDL document. These elements define the contents of the SOAP Envelope, Header, and Body.

The SOAP schema for binding defines a number of elements that can influence the way a SOAP message is formatted. This article focuses on the most common elements (and their attributes).

In Figure 10, the elements in bold are defined by the WSDL schema. The elements in italic are defined by the SOAP schema. Note that the WSDL input element, which represents the request from the client, is composed of a SOAP Header and a SOAP Body. Similarly, the WSDL output element, which is the response from the service, is bound to the SOAP Body.

```
<binding>
      <soap:binding/>
         <operation>
            <soap:operation/>
                  <input>
                     <soap:header/>
                     <soap:body/>
                  </input>

                  <output>
                     <soap:body/>
                  </output>
          </operation>
</binding>
```

Figure 10: Binding hierarchy for WSDL and SOAP

The *header* element allows the application to specify a SOAP header in the header section of the SOAP Envelope. As noted earlier, applications can specify their own headers to extend the SOAP protocol. For example, an application might add a unique header to monitor the activity of a stock broker.

The *binding* element has two attributes of interest: **transport** and **style**. The **transport** attribute specifies the transport that SOAP uses. *Note: HTTP is used in almost all cases and is the only transport supported by the WS-I Basic Profile.*

The **style** attribute defines the default SOAP messaging style to use for the entire interface (portType). The style can have one of two values: RPC or document.

RPC-style request

If RPC is specified, the SOAP Body will contain the information required to call a method on an interface. The required information is obtained from the WSDL elements. The following table indicates which WSDL elements are to be used when writing the SOAP Body for an HTTP request.

Element used	Where the data comes from in WSDL
Method name	The **name** attribute of the *operation* element
Method argument name	The **name** attribute of the *input* element

Using the previous stock quote example, the WSDL document in Figure 11 highlights the operation and input elements that will be used to build the payload of the SOAP Body. Figure 12 shows the associated SOAP message for this request.

```
<!-- message elements describe the data passed as input and output -->
<message name="getStockPriceRequest">
    <part name="symbol" type="xsd:string"> </part>
</message>

<message name="getStockPriceResponse">
    <part name="price" type="xsd:float"> </part>
</message>

<!-- portType and operation element describe the interface -->
<portType name=”StockQuote”>
  <operation name=”getStockprice”>
   <input name="symbol" message="abc:getStockPriceRequest"> </input>
   <output name="price" message="abc:getStockPriceResponse"></output>
  </operation>
</portType>
```

Figure 11: WSDL for an RPC-style request

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
xmlns:abc="http://abc.com/stock">
<soap:Body>
   <abc:getStockPrice>
      <symbol>HPQ</symbol>
   </abc:getStockPrice>
</soap:Body>
</soap:Envelope>
```

Figure 12: SOAP message for the RPC-style request

Document-style request

If document is specified, the SOAP Body will contain raw XML. The XML content is defined by the application. The WSDL shown in Figure 13 is slightly different when using document style. Notice that the part element has an attribute named **element**, which is used to include the employee XML schema that we defined earlier in this article.

```
<!-- include application defined employee schema -->
<types>
  <xsd:schema targetNamespace=http://abc.com/employee>
      <xsd:import namespace="http://abc.com/employee"
         schemaLocation="http://abc.com/employee.xsd"/>
  </xsd:schema>
</types>

<!-- message element describes the data passed as input -->
<message name="updateEmployee ">
    <part name="employeeRec" element="abc:employee"> </part>
</message>


<!-- portType and operation element describe the interface -->
<portType name="HumanResources">
  <operation name="EmployeeRecord">
   <input name="employeeRec" message="updateEmployee"> </input>
</portType>
```

Figure 13: WSDL for a document-style request

Figure 14 shows the associated document-style request. Note that the XML is inserted into the SOAP Body without any extra definition. With document-style messaging, the application can send any XML elements without SOAP dictating a structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
xmlns:abc="http://abc.com/employee">
<soap:Body>
<employee>
      <name>
        <first>David</first>
        <last>Sullivan</last>
      </name>
      <email>davidjsullivan@hp.com</email>
</employee>
</soap:Body>
</soap:Envelope>
```

Figure 14: Associated document-style SOAP request

Group 3: Define a service and give it a name

Group 3 elements create a web service for use by clients. The service element contains one or more port elements. A port element has an associated binding and an address. The address is a unique URI that is used by clients to call this web service. In Figure 15, the web service StockQuoteService is given the unique address `http://abc.com/employee/stockQuote`.

```
<service name="StockQuoteService">
  <port name="StockQuoteService_port"  binding="StockQuote_RPCbinding">
    <soap:address location="http://abc.com/employee/stockQuote" />
  </port>
</service>
```

Figure 15: WSDL defining the service to be called by clients

UDDI

UDDI stands for Universal Description Discovery and Implementation. UDDI registries act as repositories for web services. Web service providers can publish their services for others to use. Web service consumers can "shop" a UDDI registry to find the service that best fits its particular needs.

Web service providers register their business in a UDDI registry. They describe their business along with the services that they provide. A common analogy for describing the properties of a single registration is based on the United States phone book. For each registration there are three different levels of detail available. The first level is like the white pages: it provides basic information about a company, such as the name, address and phone number. The second level is like the yellow pages: it provides a categorization of a service provider and their services. A single service can appear in different categories. For instance, a category might be based on the type of service or on the service's geographical location. The third level is like the green pages: it provides the technical details of how to use the service.

A web service consumer uses the UDDI registry to locate services. Typically, the consumer can find a service in one of two ways: using a web browser and programmatically. The web interface allows a person to browse the many categories in the registry and read descriptions of the business and services. The programmatic API allows applications the ability to browse and discover businesses and their services.

UDDI registries can be public or private. Public registries are used by businesses to advertise their company and the services that they support. A private registry can be used within a company or within a division of a company to provide services to their employees.

Applications use UDDI registries to publish or discover web services. When a web service is developed, a publishing API is required to place the web service description in the registry. When a client wants to look up a web service, a discovery API is required. Some web service platforms provide implementations for both the publishing and discovery programming interfaces. Some other supply only the discovery interface and still others supply neither.

The UDDI registry is typically used at development time. If the developer already knows which web service they want to use in the application, then a UDDI registry is not needed.

OpenVMS web service products and tools

The OpenVMS integration strategy encourages partners to provide best-in-breed products whenever available. OpenVMS will continue to port key open source products to provide a range of choices, enabling users to pick the product that best fits their particular situation. Where appropriate, OpenVMS will provide OpenVMS-aware tools to aid in the development and deployment of web services. A number of web service related products have either been ported to OpenVMS or are known to work on OpenVMS. Table 4 lists these products.

Table 4: Web service related products for OpenVMS

Open source

Product	Description
SOAP Toolkit V1	Web services development and deployment toolkit. Based on the older Apache SOAP Toolkit.
SOAP Toolkit V2	Web services development and deployment toolkit. Based on Apache Axis.
XML-J	Java XML parser supporting DOM and SAX interfaces. Based on Apache Xerces. Java transformation engine for converting XML document to other HTTP and other XML documents. Based on Apache Xalan.
XML-C	C++ XML parser supporting DOM and SAX interfaces. Based on Apache Xerces. C++ transformation engine for converting XML document to other HTTP and other XML documents. Based on Apache Xalan.
Apache Ant	Build environment supplied as part of Apache Tomcat.
Apache Log4j	Java logging facility used for testing and debugging.
NetBeans	Integrated Java development environment with XML support.

Partner products

Product	Description
BEA WebLogic Server (WLS)	Enterprise grade J2EE application server.

HP products

Product	Description
HP BridgeWorks	OpenVMS aware application integration middleware.

Providing web services for OpenVMS applications

The majority of products listed in Table 4 allow OpenVMS applications written in Java to easily create and consume web services. However, most OpenVMS applications are not implemented in Java. For these applications, more development time is required to wrap their application and expose them as web service. OpenVMS engineering understands the importance of web services for the long-term viability of non-Java applications and intends to provide customers with tools that will simplify the creation of web services for non-Java applications.

In the same way that vendors provide tools to generate web services wrappers from Java applications, OpenVMS intends to provide a web services toolkit to generate web services wrappers from non-Java applications.

In order to understand which components OpenVMS will supply, let's examine the different components of a web services platform. The delivery of web services features can be separated into three distinct categories. Each of these categories is needed to support web services on OpenVMS. Figure 16 illustrates the OpenVMS web services strategy.

- **A web services client to call a web service.** A client is capable of locating and calling a web service. Clients allow OpenVMS applications to reuse services on other platforms. OpenVMS relies on partners and open source projects for supporting web service clients

- **A web services broker to deploy services.** A broker takes an object (usually Java) and makes it available as a web service. OpenVMS relies on partners and open source projects for supporting web service brokers.

- **A bridge from the broker (usually Java) to the legacy application (never Java).** This is a key component in supporting existing applications. It provides the glue between the web service and the existing 3GL application. Without the bridge, only applications written in Java can be easily exposed as services. No partners or open source projects provide this component on OpenVMS. Therefore, OpenVMS will provide both development and run-time components for wrapping 3GL applications. These components will be provided as a toolkit.

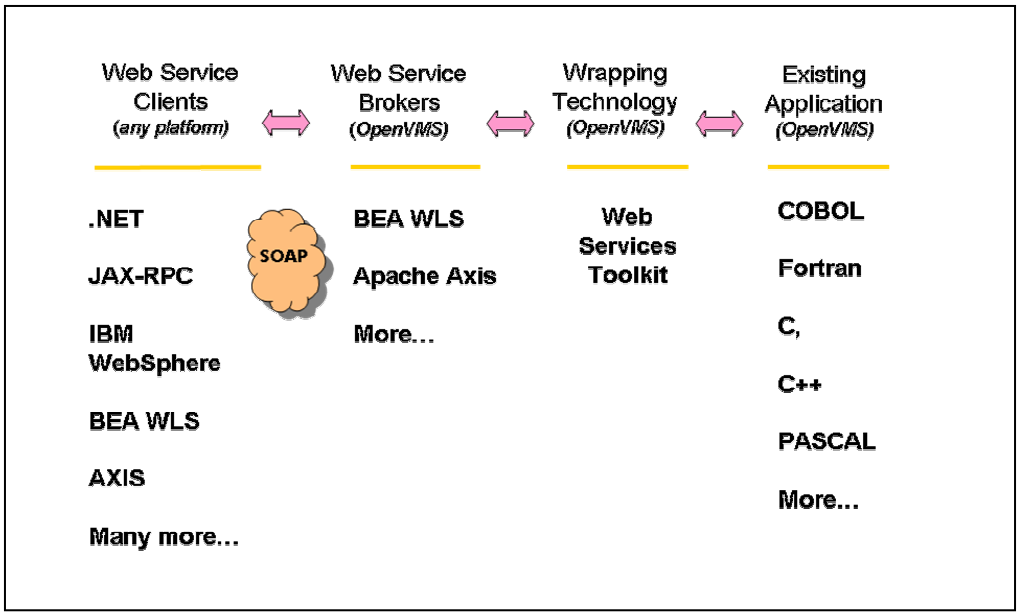

Figure 16: OpenVMS web services strategy

Recommendations

OpenVMS provides a number of products and tools for developing web services. If you have not already begun to evaluate web services, this is the time to do it. Start evaluating the RPC-style and document-style web services designs. Identify the areas in your organization where web services would provide a value.

It is important to remember that web service development platforms can generate code on your behalf. These tools handle the details of WSDL and SOAP. Now that you have an understanding of these technologies, you can alter them as needed to address your unique situation.

Most of the details of the web service technologies can be learned as needed. Most of the key concepts of web services are not new. They have been used in traditional middleware products for decades. You likely already understand more about web services concepts than you think.

Finally, look at the examples in the next section to see how easy it can be to create a web service on OpenVMS and call it from client implementations such as .NET and JAX-RPC.

Web services examples

As mentioned earlier, developing a web service is not difficult. To illustrate how easy using web services can be, the examples do the following:

- Create a very simple OpenVMS web service.
- Test the services from a web browser.
- Review Java code that uses the JAX-RPC interface to call the OpenVMS service.
- Review C# code that uses Microsoft .NET to call the OpenVMS service.

After you see how easy this process can be, play with the service to make it more interesting. The examples source code is available at the URI:

`http://h71000.www7.hp.com/openvms/products/ips/soap/webservices.jar`

To extract the contents of the jar file, issue the following command:

```
jar xvf webservices.jar
```

Install the required software

First, install the products listed in Table 5.

Table 5: Required software for OpenVMS development environment

Target platform	OpenVMS V7.2-2 or higher
Target language	Java
Required software	Java V1.3.1 or higher `http://h18012.www1.hp.com/java/download/index.html`
	CSWS_JAVA (Apache Tomcat) `http://h71000.www7.hp.com/openvms/products/ips/apache/csws_java.html`
	SOAP_Toolkit V2.0 (Apache Axis) `http://h71000.www7.hp.com/openvms/products/ips/soap/soap.html`

Configure the required software

Verify that each product is installed properly and that all logicals are defined. Run the product's installation verification procedure (IVP), if available.

Step 1:

After you install the SOAP Toolkit V2.0, run the configuration utility located in the AXIS$ROOT:[AXIS-1_1.OPENVMS.COMS]SOAP_TOOLKIT-2_0_UTIL.COM directory, and select "Copy supplemental jar files to AXIS$ROOT:[AXIS-1_1.LIB]". For example:

```
$ axis$root:[axis-1_1.openvms.coms]soap_toolkit-2_0_util

           SOAP Toolkit 2.0 (based on Apache AXIS) Utility
           -----------------------------------------------
        1 - Show current configuration

        2 - Run SOAP Toolkit 2.0 Validation Procedure

        3 - Copy supplemental jar files to AXIS$ROOT:[AXIS-1_1.LIB]

        E. Exit SOAP Toolkit 2.0 Utility
           -----------------------------------------------
Please choose a task: 3

issuing command: copy/log AXIS$ROOT:[AXIS-1_1.LIBJARS] AXIS$ROOT:[AXIS-1_1.LIB]
/exclude=(xercesImpl.jar,xml-apis.jar)

%COPY-S-COPIED, AXIS$ROOT:[AXIS-1_1.openvms.libjars]activation.jar;1 copied to
AXIS$ROOT:[AXIS-1_1.LIB]activation.jar;1 (107 blocks)
%COPY-S-COPIED, AXIS$ROOT:[AXIS-1_1.openvms.libjars]ant.jar;1 copied to
AXIS$ROOT:[AXIS-1_1.LIB]ant.jar;1 (1440 blocks)
%COPY-S-COPIED, AXIS$ROOT:[AXIS-1_1.openvms.libjars]junit.jar;1 copied to
AXIS$ROOT:[AXIS-1_1.LIB]junit.jar;1 (237 blocks)
%COPY-S-COPIED, AXIS$ROOT:[AXIS-1_1.openvms.libjars]optional.jar;1 copied to
AXIS$ROOT:[AXIS-1_1.LIB]optional.jar;1 (1315 blocks)
%COPY-S-NEWFILES, 4 files created

Choice  (R- Return to Menu)  (E- Exit):
```

Step 2:

Configure Axis to run under Tomcat by copying the Axis webapps subdirectory under the Tomcat webapps subdirectory. To determine the Tomcat home directory, run the SYS$STARTUP:APACHE$JAKARTA.COM command procedure.

For example, if the Tomcat home directory is DISK$:[CSWS_JAVA.APACHE.JAKARTA.TOMCAT], use the following backup command:

```
$ BACKUP/LOG/IGNORE=INTERLOCK AXIS$ROOT:[AXIS-1_1.WEBAPPS...]*.* -
_$ DISK$:[CSWS_JAVA.APACHE.JAKARTA.TOMCAT.WEBAPPS]
```

Step 3:

Start Tomcat by running the SYS$STARTUP:APACHE$JAKARTA.COM command procedure.

Create a simple web service

We will keep the web service as simple as possible. The service simply returns the IP address of the OpenVMS system. From any directory, type or copy the following code into a file named HELLOOPENVMS.JAVA.

```
import java.net.*;

public class helloOpenVMS {

    /** Creates a new instance of helloOpenVMS */
    public helloOpenVMS() {
    }

    public String getAddress() {

        try {

            return InetAddress.getLocalHost().getHostAddress();

        } catch (java.net.UnknownHostException e) {
            return "could not retrieve host IP address";
        }
    }
}
```

Deploy the web service

Step 1:

Verify that JAVA$CLASSPATH is defined and that Tomcat is started.

Step 2:

Rename the file a file extension of .JWS (Java web service).

```
$ RENAME HELLOOPENVMS.JAVA HELLOOPENVMS.JWS
```

Step 3:

Copy the HELLOOPENVMS.JWS file to the Axis webapps deployment directory under the Tomcat home directory.

For example, if the Tomcat home directory is DISK$:[CSWS_JAVA.APACHE.JAKARTA.TOMCAT], use the following command:

```
$ COPY HELLOOPENVMS.JWS -
_$ DISK$:[CSWS_JAVA.APACHE.JAKARTA.TOMCAT.WEBAPPS.AXIS]
```

Congratulations! Your web service is ready to be tested.

Test the web service from a browser

Since we have not yet written a web services client, we will use a web browser to test our new service. The browser can be located on any computer that has access to the computer where the web service is deployed.

Type or copy the following URL into the address window of your favorite browser. Make sure you replace *your-computer-name* with the name of the OpenVMS computer that deployed your web service.

`http://your-computer-name:8080/axis/helloOpenVMS.jws?method=getAddress`

This URL calls the web service broker located on *your-computer-name*. The broker is listening on port 8080 for incoming SOAP requests, such as this one. The web service is located in the Axis directory. The name of the web service is helloOpenVMS.jws. We want to call the method getAddress, so we specify this by using the syntax ?method=getAddress.

Look at the SOAP Response

After you send the request to the web service, you should see something like the following text returned in your browser. This is the SOAP response from your web service.

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
<soapenv:Body>
  <getAddressResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <getAddressReturn xsi:type="xsd:string">16.32.128.78</getAddressReturn>
  </getAddressResponse>
</soapenv:Body>
</soapenv:Envelope>
```

It is very easy to understand the SOAP response from the web service. Notice that the SOAP Body contains our method getAddressResponse and specifies that the method returned the address 16.32.128.78. (This address will be different for your system.)

Look at the WSDL

Let's look at the WSDL generated by Axis for our web service. Type or copy the following URL into the address window of your favorite browser. Make sure you replace *your-computer-name* with the name of the OpenVMS computer that deployed your web service.

`http://your-computer-name:8080/axis/helloOpenVMS.jws?wsdl`

This is the WSDL generated for the preceding SOAP response:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://16.32.128.78:8080/helloOpenVMS.jws"
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:apachesoap="http://xml.apache.org/xml-
  soap" xmlns:impl="http://16.32.128.78:8080/helloOpenVMS.jws"
  xmlns:intf="http://16.32.128.78:8080/helloOpenVMS.jws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:message name="getAddressResponse">
    <wsdl:part name="getAddressReturn" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="getAddressRequest">
</wsdl:message>

<wsdl:portType name="helloOpenVMS">
    <wsdl:operation name="getAddress">
        <wsdl:input message="impl:getAddressRequest"   name="getAddressRequest"/>
        <wsdl:output message="impl:getAddressResponse" name="getAddressResponse"/>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="helloOpenVMSSoapBinding" type="impl:helloOpenVMS">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="getAddress">
      <wsdlsoap:operation soapAction=""/>

      <wsdl:input name="getAddressRequest">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://DefaultNamespace" use="encoded"/>
      </wsdl:input>

      <wsdl:output name="getAddressResponse">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://16.32.128.78:8080/helloOpenVMS.jws" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>

</wsdl:binding>


  <wsdl:service name="helloOpenVMSService">
    <wsdl:port binding="impl:helloOpenVMSSoapBinding" name="helloOpenVMS">
      <wsdlsoap:address location="http://16.32.128.78:8080/helloOpenVMS.jws"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

In this WSDL output, the *message* element is named getAddressResponse. It has a *part* element named getAddressReturn, which is a string returned from the service -- in this case, an IP address.

Writing a simple client using JAX-RPC

Now let's look at the client-side Java code that uses the JAX-RPC interface to call our service. If you want to create this client yourself, refer to the detailed instructions in the samples download. For this client:

- Target platform: any platform that supports Java (Linux, UNIX, Windows, OpenVMS)
- Target language: Java

The following Java code uses the JAX-RPC interface to call the OpenVMS web service The preceding few lines of code are an entire web services client. In this example, the client code establishes two JAX-RPC objects, Service and Call. The address of the helloOpenVMS web service is set and the method getAddress is invoked. It's all very simple.

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;

public class getAddress {

    /** Creates a new instance of getAddress */
    public getAddress() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        printAddress();
    }

    static void printAddress(){
       try {

            String endpoint =
                 "http://yourcomputername:8080/axis/helloOpenVMS.jws";

            Service service = new Service();

            Call call = (Call)service.createCall();
            call.setTargetEndpointAddress( new java.net.URL(endpoint) );

            String ret = (String) call.invoke( "getAddress" , null);

            System.out.println("got IP address: " + ret);

           } catch (Exception e) {
                 System.err.println(e.toString());
           }
    }
}
```

Note: JAX-RPC is one of a number of Java interfaces for web services.

Writing a simple client using Microsoft .NET

Now let's look at the client-side C# (C Sharp) code that uses the .NET environment to call our service. If you want to create this client yourself, refer to the detailed instructions in the samples download. For this client:

- Target platform: any Windows platform that supports .NET
- Target language: C#

The following C# code uses the .NET environment to call the OpenVMS web service. This code is also very simple. A .NET web reference is generated using a simple wizard. This web reference generates client-side proxy code that hides details of the invocation of the OpenVMS web service.

For a description of how to create a web reference, refer to the detailed instructions in the samples download.

```
using System;
using webreference.proxy;
namespace addressPicker
{

  class Class1
  {

     /// The main entry point for the application.
     [STAThread]
      static void Main(string[] args)
      {
         helloOpenVMSService myserv = new helloOpenVMSService();

         String ret = myserv.getAddress();

         System.Console.WriteLine("the Service returned the string " + ret);
      }
   }
}
```

Summary

Web services provide both a tactical and strategic advantage to businesses. Simplicity, low cost, and high cross-platform availability have led to an unprecedented acceptance of web services by enterprise software vendors. OpenVMS customers can use web services to reduce costs while increasing the speed and quality of development projects.

OpenVMS provides best-in-breed products to its customers. With the Web Services Toolkit, applications have new and unique opportunities to leverage OpenVMS strengths from other platforms, such as Microsoft .NET and J2EE.

For more information

The examples used in this paper are available from the following location:

http://h71000.www7.hp.com/openvms/products/ips/soap/webservices.jar

Feel free to contact the author of this paper by sending email to **davdjsullivan@hp.com**.