View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Clute Institute: Journals

Foundations Of Service Science Technology And Architecture

Harry Katzan, Jr., Savannah State University, USA

ABSTRACT

This paper concludes the conspectus of Service Science for academicians and practitioners. It follows the two previous papers, entitled Foundations of Service Science: Concepts and Facilities and Foundations of Service Science: Management and Business, with the express purpose of defining the scope of the discipline. An eclectic background in service technology and service architecture is required to fully explore the research potential of a science based on services. This paper reviews the technical concepts needed to apply the concepts that have previously been introduced.

Keywords: Service technology, messaging, message patterns, message topology, mail model, Web services, service-oriented architecture.

SERVICE TECHNOLOGY CONCEPTS

The basis of service technology is really straightforward. Clients and providers communicate with one another through the use of messages and contracts, and in many areas of service, the communication involves information and communications technology (ICT). A client and a provider can be tightly coupled, as when a patient is sitting in front of the doctor and they are having a give-and-take conversation, or loosely coupled, as when you send a request to someone via email and receive a response at some undetermined time in the future. In the former case, the client and provider are communicating in a *synchronous mode* without technology, and in the later case, they are communicating in an *asynchronous mode* with the use of technology. The *contract* is a formal or informal agreement that delineates the service in which the client and provider are engaged. The contract can be a formal document, an informal agreement, or be implicit in the activity under consideration. Another view of a contract is that it is a specification of how to use a service and what to expect from a service.

Messaging Basics

Each service event requires at least one message, and each message requires a context, which gives meaning to the interaction. Entities that participate in a service-oriented message are called the message sender, the message intermediary, and the message receiver. When you fire up your Internet browser, for example, and enter a World Wide Web address, such as www.ibm.com, the browser sends a message to the IBM server somewhere out in cyberspace. The browser, acting on your behalf as the client of the Web service, is the message sender. The IBM web site is the message receiver. When IBM sends its home page back to be rendered for you by your browser, the roles are reversed; it is the sender and your browser is the receiver, and the Internet is the message intermediary.

The *message* is the glue that ties a service together.

Conceptual Model of Service Processing

The most profound aspect of service science is that a *service is a process*, as suggested by the following message pattern:

1. A client sends a message to a service provider.
2. The provider performs the required action and returns a message to the client.

The focus is on the data that is transmitted and not on the communications medium, which can take the form of a human interaction or a computer-based message. The context for the message can be embedded in the message or it can be inherent in the way that the service provider is addressed. The importance of context is suggested by the cartoon floating around where two dogs are seated in front of a computer screen. One dog says to the other, "On the Internet, no one knows you're a dog." Two good rules of thumb are that in face-to-face services, interpersonal communication provides the context. In human-to-computer services, the context must be inherent in the message. For example, entering "Boston Red Sox" into your browser to get the score of the last World Series game is probably going to generate a lot of miscellaneous information in which you are not interested, because you provided no context.

Initially, it is useful to recognize that we are operating at two levels: the service level and the message level. At the *service level*, the message entity that receives the message is the service provider and is regarded simply as the **service**. At the *message level*, there is some choreography involved with providing a service, as demonstrated by the above two-step interaction. In fact, a service may involve the interchange of several messages.

Enterprise Service Technology

Many modern enterprises (i.e., business, government, education) provide computer support to internal users, clients, business partners, and other enterprise entities. The facilities are usually integrated into administrative, product development, supply chain, or customer relationship operations. Because those services, consisting of computer applications and associated procedures, are tried, tested, and dependable, it would be prudent to use them as building blocks for new enterprise applications.

The concept that underlies service orientation is that it is simply more efficient and reliable to identify the bundled services and package them as reusable components than it would be to rewrite them. Bundled services could then be used by other services, so that information system applications could be developed more rapidly and enable the enterprise to be more responsive to external conditions. This practice is the basis of web services that are covered in this paper.

A typical business function that lends itself to componentization is to perform a credit check on a prospective customer before confirming a large order. Such a check is normally performed in different operational systems in an enterprise. After restructuring, the credit check software is packaged as a single business component and exposed as an enterprise service for use by other enterprise service applications.

SERVICE MESSAGING

When two service entities are engaged in communication, they are regarded as being *connected*. An enterprise has two options for developing a service connection:

1. Message entity to message entity (ME → ME)
2. Message entity to enterprise entity (ME →EE)

The first option, denoted by ME → ME, refers to either a client-to-provider or a provider-to-client communication. The second option denoted by ME →EE refers to a client-to-many-provider communication. The notion of connectedness is needed for an appreciation of message patterns and topologies, covered in the next section. For example, the ME →ME option may represent the case where an order-processing application sends a message to a shipping application to have an item shipped to a customer. The ME →EE option might represent the case where an airline's flight operations application sends a message to other involved computer applications, such as scheduling and reservations, when a plane has taken off.

Message Patterns

A *message pattern* is a model of service communications that represents a single connection between one sender and one receiver. There are three basic patterns representing message traffic that can go only one way, both ways but only one way at a time, and both ways simultaneously.

The one-way message flow is regarded as a "fire-and-forget-it" send, also known as *simplex* and *datagram* communications service in the computer community. The second model is the request/reply model, known as *half duplex*, wherein only one participant communicates at a time as with the walkie-talkie type of interaction. In the final model, called *full duplex*, both messaging participants can send messages at the same time, as in an ordinary telephone conversation. Clearly, messaging can take on different patterns depending upon the operational environment used for technical support.

Message Structure

In its most simple form, a message is a string of characters encoded using standardized coding methods commonly employed in computer and information technology. Messages have a uniform format consisting of a header and a body. The *header* primarily concerns addressing and includes the addresses of the sender and the receiver. In the request/reply message pattern, the return address is picked up from the message header for the response portion of the transaction. The *body* of the message contains the information content of the message, and because it is intended only for the receiver, is not usually regarded during message transmission.

The manner in which messages are structured is similar to the way that letters are handled by the postal service. The outside of the envelope contains addressing information and the insides are handled as private information.

Message Topology

Message topology refers to the manner in which messages are sent between messaging participants, and not necessarily to the communication techniques used to send them. The most widely used form of communication is known as *point-to-point* using any of the message patterns given above. Usually, point-to-point implies the request/reply message pattern where the reply address is picked up from the message header. A variation to point-to-point is *forward-only point-to-point* where a message reply is not expected.

Message Interactions

In most cases of messaging, the sending participant needs to know that the receiving participant is listening before the real message is transmitted. It is something like the following:

Sender: Are you listening?
Receiver: Yes.
Sender: Are you Gregory Charles Cabot?
Receiver: Yes.
Sender: You've just won one million dollars.

OK, it's a bit contrived and also, it's messaging at the service level. There is also handshaking going on at the message level, which we are going to cover in the next section.

The following example demonstrates message interaction through instant messaging at the service level. It demonstrates message interactions.[1] For this instance, **User A** is sending an instant message to **User B** who responds to **User A.** The interaction consists of four distinct messages, delineated as follows:

[1] This example is adapted from Van Slyke and Bélanger (2003), p. 110.

Message 1: User A logs on to the instant messaging (IM) server. The expected response is that the IM server will return a message with the users in A's group that are currently logged on. The message goes from User A through the Internet to the IM server.

Message 2: The IM server sends a message to User A with the members that are logged on. The message goes from the IM server to User A.

Message 3 : User A sends a message, such as "Hi User B," to User B. The message goes from User A through the Internet to the IM server. The IM server then sends the message through the Internet to User B.

Message 4: User B responds with a message, such as "Hi yourself," to User A. The message goes through the Internet to the IM server. Then the IM server sends the message through the Internet to User A.

Most people would regard this interaction sequence in which User A sends an instant message to User B as a service and B's response to A as another service. Popping up a level, the service provider is the instant messaging server and users A and B are clients.

SERVICES ON THE INTERNET AND THE WORLD WIDE WEB

A service that takes place on the Internet and the World Wide Web is called a *web service*.[2] A web service is a process in which the provider and client interact to produce a value; it is a pure service. The only difference between a web service and medical provisioning, for example, is that in the former case, the client and provider are computer systems. Ordinary email is a web service. Requesting a home page from a provider's web site is a web service. Sending an instant message over the Internet is a web service. Almost anything you can think of doing on the web would be called a web service. However, there is another category of service known as a Web Service. Note that Web Service is a proper noun. It is a formal process, developed by organizations such as Microsoft, IBM, and others, for conducting business over the Internet. It is covered separately.

Simple Mail Model

The most pervasive web service computer application on the Internet is electronic mail, commonly known as email. It is used in two ways: (1) To communicate between email clients; and (2) To provide a record that communication has taken place – or at least, to show that an attempt at communication has taken place. Clearly, email is designed to be a person-to-person endeavor.[3] There are two scenarios that are relevant to web services.

In the first scenario, we have a desktop personal computer (PC) operating as an email client – referred to as a PC running an email client – from which the end user sends and receives email. The email client is connected to incoming and outgoing email servers through a local-area network or a dial-up, broadband cable, or DSL connection to an Internet service provider (ISP) that is in turn connected to the mail servers. Email messages are normally managed locally, which means they are downloaded and stored on the end user's computer. When the end user decides to access email messages, he or she presses a receive button and incoming messages, stored on the incoming email server, are transferred to the local email client. Similarly, when the end user constructs a message for sending, a send key is pressed to transfer it to the outgoing email server for subsequent forwarding over the Internet. An email client uses push technology to send email messages and pull technology to receive email messages.

In the second scenario, we again have an email client for message management. The email client, however, is connected to an email-service server via the Internet through a local browser. The service access point is an

[2] There is another definition for web services that is slightly more specific. Cerami (2002) states, "A web service is any service that is available over the Internet, uses a standardized XML messaging system, and is not tied to one operating system or programming language."

[3] Email is a person-to–person construct but requires a slight interpretation. A sender can send a message to a mailing list; however, each individual *send operation* is still a person-to-person operation.

account set up on an Internet service portal. A web based email account is used in the same manner as in the local scenario, except that the email server is remote.

The concept of remote service server is also a platform for other web applications, such as word processing and spreadsheet operations. A remote service server that provides application functionality is known as an *application service provider*, and exists as an alternative to purchasing infrequently used software.

Service Model for the World Wide Web

When addressing a web service, there is a certain way that most people go about doing things. It's not entirely clear whether the web service architecture determines how people use the web or, the other way around, whether the architecture of the web reflects how people use it. We are calling it a generic web services model.

Imagine the following scenario. You're interested in purchasing a pair of running shoes and don't know any brands or web sites. So what do you do? You point your browser to a search engine, such as Google™, enter the words "running shoes" in the search window and click the "search" button or press the enter key. Your message is sent to Google's web server that searches an index of key words, created beforehand, and makes a list of appropriate web sites, just for you. The web server then prepares your list in a language called HTML and sends it back to your browser over the Internet. The browser then renders the HTML statements into a readable form. This is an example of the first element in the web services model. It's known as *discovery*. The service process was accomplished without regard to time, distance, or the kind of hardware and software.

Each of the entries (known as a "hit") on the resultant running-shoe list gives a brief description and a *hyperlink* with which to obtain more definitive information. This process reflects the second element in the web services model, and it is known as *description*. Various enterprises have web sites and associated web pages containing descriptive information of interest. In a separate operation, the organization behind the search engine searches the web sites in cyberspace and prepares indexes for fast retrieval.

If your goal is information, then this is perhaps as far as you will go with this example. If you are going to make a purchase over the web using an appropriate site, then the next step is to *bind* to that web site and go through an interactive process for selection, payment, and delivery. Each step in the bind process requires additional web services, so that a web service is essentially a cascading series of other web services.

A variety of tools and techniques are required for a successful implementation of web services architecture. Whenever there is a service, there is communication; and whenever there is communication, there are messages. Whenever there is a message, there is a context so that the intent of the service can be sustained. These elements are present in one form or another in all services, ranging from the more straightforward human interaction to the operation of a sophisticated enterprise computer application.

HyperText Transfer Protocol

HyperText Transfer Protocol (HTTP) is a collection of rules and procedures for transferring messages between computers over the World Wide Web. Without HTTP, the web would not be the revolutionary phenomena that it is today. When you make a service request over the web, your entry goes through your browser before it goes over the Internet. Here's how.

When you fire up your browser, you are initiating the execution of a program that runs on your personal computer, workstation, personal digital assistant (PDA), cell phone, terminal – or whatever you choose to use. Now that computing device is performing a service for you in the sense that you can now do things you could not possibly do without it. In fact, you could run all manner of programs, such as productivity software that does word processing, without any information leaving or entering your local environment. As far as the Internet is concerned,

however, essentially nothing has happened. You type a URL[4] into your browser window and press the enter key, and then things start to happen. This is when HTTP gets into the act.

Your browser prepares a message called a HTTP request, such as[5]

```
GET /index.html HTTP/1.1
Host: www.example.com
```

and sends it over the Internet to the web server of the "example.com" web site somewhere out in cyberspace. The web server responds in turn to the return address obtained from your message header with

```
HTTP/1.1 200 OK
Date: Mon, 02 Dec 2007 12:38:34 GMT
Server: Apache/1.3.27 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 155
Connection: close
Content-Type: text/html; charset=UTF-8
```

This response message is followed by a blank line and then the requested information that represents the contents of the file (usually the default file, such as index.html) from the site specified in the HTTP get request. The information content of the response message might take the form (highly unlikely but possible):

```
<html>
        <head>
        <title>Hello World</title>
</head>
        <body>
        <h1>Hello World</h1>
        <par>
        Greetings from Cyberspace
        </par>
        </body>
</html>
```

that would be rendered by your browser and displayed on your screen. The text is transmitted in a well-known language peculiar to the web and known as HyperText Markup Language (HTML). It is introduced in the next section.

The *HyperText Transfer Protocol* has additional verbs, such as POST, PUT, and DELETE, that facilitate the transfer of messages between a client computer and a server computer.

HYPERTEXT MARKUP LANGUAGE

Aside from the Internet information super highway and the idea of linking information pages together (i.e., the World Wide Web), probably the coolest thing that has ever happened in the over-hyped world of computers is the realization that it is possible to send a document from one computer to another and have that document displayed on the receiving end in a reasonable form without regard to the brand and model of computer, kind of software, time of day, and location. This amazing feat – and it is truly that – is possible because of hypertext markup language

[4] URL stands for Uniform Resource Locator, such as www.example.com.
[5] The examples are adapted from Wikipedia (see selected reading).

(HTML), as introduced in the previous section. We are interested in HTML for two important reasons. First, it is a useful thing to know something about, as long as you don't get hung up in the details. Secondly, HTML is a forerunner to Extensible Markup Language (XML) that is a technology for sending messages between services.

HTML Documents

To start off, an HTML document is nothing more than a bunch of characters that someone has entered into a text editor or a word processor and been saved as a file on a computer employed as a web server. When you request an answer from a web site, such as the one and only www.ibm.com, the corresponding web server goes to a default file named index.html, retrieves the HTML file, and sends it back to your browser for rendering on your computer's display. It is someone's job to put the right stuff into index.html, and that stuff should be written in HTML. Now the file named index.html might have links to other pages that are returned in a similar manner when you click on them. Those links are referred to as *hot links*, because we get some action when we click on them – as we just mentioned. You can even put programs into an HTML document. These programs are executed by your browser resulting in some visual or audio activity on the receiving end. The active behavior can result in a wide variety of audio, video, and data-oriented interactive forms.

Tags

The basis of an HTML is a tag, such as <html>, that provides information to the receiving browser. In the case of <html>, for example, the tag indicates the beginning of an HTML document. Actually, a tag is only a strong suggestion, since each browser has a mind of its own. Most tags have an enclosing tag, such as </html>, that delineates a section of a document, such as in the following HTML snippet:

```
<html>
    <head>
        <title>University of the United States</title>
    </head>
    <body>
        ●
        ●   ←--- The content goes here
        ●
    </body>
</html>
```

Tags give an HTML document structure and information on page rendering; they do not give meaning. We will use XML for that.

Discovery

One of the key aspects of web page design is to facilitate discovery whereby clients can find services. Search engine companies use a technique known as "web crawling" in which a program called a *web crawler* or a *bot* (for robot) crawls through web pages following hyperlinks to build indexes for subsequent search operations. Without additional information, all words in a web page are treated the same. You can add additional information to the "head" section to increase the fidelity of searching and increase the chances that a user will navigate to your web site.

This is where the <meta> tag comes in. With the meta tag, web page designers commonly supply three types of descriptive items: a list of keywords, a description, and the name of the web page owner – sometimes the name of an organization and sometimes an author. Search bots use this information when building indexes. The following example depicts the use of meta tags:

```
<html>
    <head>
        <title>Savannah Motor Works</title>
        <meta name=”keywords” content=”Porsche, Mercedes, BMW”>
        <meta name=”description” content=”The south’s most prestigious
                        performance car dealership”>
        <meta name=”author” content=”Gregory Cabot”>
    </head>
    <body>
        •
        •
        •
    </body>
</html>
```

Actually, there are no predefined meta tags in HTML, so a web page designer can create them to satisfy a particular need. The meta tag demonstrates a tag without an enclosing tag.

Document Elements

The HTML language has an extensive vocabulary that is a subject in its own right. A brief subset of HTML features is covered here as a forerunner to Extensible Markup Language (XML) that is used to construct messages between clients and service providers. Some of the most commonly used document elements are <h1> through <h6>, <p>, <b>, <i>, <br>, and <hr>, which represent headings, paragraph, bold face, italics, blank line, and horizontal rule, respectively. Several of these elements are depicted in the following script:

```
<html>
    <head>
        <title>My First Novel</title>
    </head>
    <body bgcolor="yellow">
        <h1 align="center">The Car</h1>
        <p align="center"> <i>by</i> </p>
        <p align="center"> <b>Gregory Cabot</b></p>
        <p> My uncle gave me my first car. It was a 1939 Chevy with fluid
         drive.  It had a flat tire and the brakes didn’t work.  It also had a
         broken window.
        </p>
        <p> My father taught me how to do the repairs and I had to do them.
          Afterwards, I didn’t like the car and sold it for $50.
        </p>
        <hr>
        <p align="center">The End</p>
    </body>
</html>
```

Of course, complete comprehension is not necessary or even expected. However, the key point has been made that HTML is a powerful tool in the construction and communication of web-based documents.

EXTENSIBLE MARKUP LANGUAGE

To put the virtues of HTML and XML into perspective, we can properly say that HTML is used to describe web pages and XML is used to describe information. XML stands for eXtensible Markup Language. Both languages use markup, a term that ostensibly is intended to imply that someone prepares a document and then

incorporates descriptive elements to suggest how the document should look when displayed *or* to communicate the intended meaning of the document. With XML, markup gives semantic information as suggested by the following script:

```
<?xml version="1.0" ?>
<library>
    <library_name>Pleasure Books</ library_name>
    <book>
        <title>The DaVinci Code</title>
        <author>Dan Brown</author>
    </book>
    <book>
        <title>The Secret Servant</title>
        <author>Daniel Silva</author>
    </book>
</library>
```

We will call the semantic information "tags" as we did with HTML, even though XML specialists refer to them as "element type names." An XML document must contain a prolog and at least one enclosing document element. In the above example, the following statement is the prolog:

<?xml version="1.0" ?>and the enclosing document element is:

```
<library>
    •
    •
    •
</library>
```

This is an example of a main element that must be present in all XML documents. It is often referred to as the *root element*, and it is the characteristic that gives an XML document a hierarchical structure. All opening tags in XML, such as <book>, must have closing tags, such as </book>. With XML, we can make up our own tags, since we are using the language to describe information that has a specific meaning.

Rendering an XML Document

Even though an XML document, by definition, is intended for communication, we can display the contents in a particular form by using a stylesheet. To use a stylesheet, we have to extend the prolog with a statement of the form:

```
<?xml:stylesheet href="library.css" type="text/css" ?>
```

and develop a stylesheet description file, named library.css in this example, that would have descriptive content, such as the following:

```
library_name {
    display: block;
    font: bold 24pt;
}
title {
    margin-top: 20px;
    display: block;
    font: italic 18pt;
}
author {
    display: block;
    font: 12pt;
}
```

A rendering of the penultimate XML document would be achieved with the library.css stylesheet file**.**

Additional XML Features

There is a lot more to the XML language, such as a formal means of defining data types and stylistic structures for XML documents along with a whole host of operational facilities. If it would take one book to totally describe HTML, it would take two books to fully give the features in XML. For a basic knowledge of service science, complete comprehension of XML is not needed – only an idea of what it is all about.

At this point, we have enough knowledge of service tools and techniques to proceed with Web Services, introduced in the next section. We are going to start with a specification of the XML grammar for a form of web messaging known as SOAP, which was initially an acronym for Simple Object Access Protocol.

WEB SERVICES

A Web Service has been defined as any service that is available over the Internet, uses a standard XML messaging system, and is not dependant upon any one particular operating system.[6] This statement has the makings of something different from the "web service" that was presented earlier when discussing HTTP. Well, it is. Earlier, we described the human web wherein an end-user sends an informational request via HTTP to a web server, and the requested information is returned, also via HTTP, to the user's browser for visual display. In this section, we are going to cover the automated web, in which one computer sends information in the form of an XML document to another computer over the Internet, and the intended result is to initiate a service of some kind. The latter form is a well-defined web service model such that the name Web Service is a proper noun. It is important to mention that XML is used for things other than Web Services. In just so happens that they grew up together, so that they are naturally associated with one another.

Web Service Concepts

There are two general approaches to using a Web Service. The first is to have one computer (*the sender)* send a simple message to a second computer (*the receiver*) to have the receiver execute a procedure for the sender and return the result. The procedure is known as a *method* and the process is referred to as an XML-RPC, which stands for *XML-Remote Procedure Call*. A frequently used example to demonstrate the concept is the weather service application: a requester sends a zip code to the weather service program and the program (i.e., the method) returns the temperature. The initial request message can be written in XML as:

```
<?xml version="1.0"?>
<weatherRPC>
    <weatherMethod>getTemperature</weatherMethod>
    <parameters>
        <zip_code>29909</zip_code>
    </ parameters >
</weatherRPC>
```

The example is conceptual and the message headers and other information are omitted. The response from the weather service would take the form:

```
<?xml version="1.0"?>
<weatherResponse>
    <parameters>
        <value><int>75</int></value>
    </parameters>
</weatherResponse>
```

[6]See E. Cerami, *Web Services Essentials* (Selected Reading), for much of the subject matter in this section.

XML-RPC can be implemented via an HTTP request/response or by embedding the informational content of the transaction in a SOAP message, which is the second approach.

SOAP is a protocol for exchanging information between computers where the structure of the information is represented in XML. The basic idea underlying SOAP messaging is to make sure that programs running on two communicating computer platforms have the capability of understanding each other. Accordingly, the XML element definitions from several namespaces need to be specified in a standard manner and also be accessible over the Internet. We are not going to include the definitions, per se, in the SOAP message, but instead, include a reference to the definitions, so that if things change, every message in the world does not have to change.

A simplex (i.e., one-way) message from a client to a server or from a server to a client is called a *SOAP message* and consists of a SOAP envelope in which is placed a message header and a message body. The optional header is intended to allow the inclusion of application-specific information, such as security and account numbers. The required message body contains the references and informational content of the SOAP message. Here is what the SOAP envelope looks like:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”
    xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
    xmlns:xsd=”http://www.w3.org/2001/XMLSchema”>
    ●
    ●   ←----- The message body would go here
    ●
</SOAP-ENV:Envelope>
```

and a sample message body is

```
<SOAP-ENV:Body>
    <ns1:getTemp
    xmlns:ns1=”urn:xmethods-Temperature”
    SOAP-ENV:encodingStyle=”http://schemas.xmlsoap.org/soap/encoding/”>
        <zipcode xsi:type=”xsd:string”>29909</zipcode>
    </ns1:getTemp>
</SOAP-ENV:Body>
```

Again, comprehension is not required or expected. The scripts are exceedingly detailed, but one point is clear, even from this simple example. Once the structure of SOAP messaging is developed, the addition of the content can be quite straightforward.

To sum up this section, SOAP messaging is straightforward. All one SOAP client has to do to send a message to another SOAP client is to put the content document into a predetermined SOAP message structure and send it through the Internet. It doesn’t matter if the sender is a client or a server, as long as the sender and the receiver are both SOAP clients. The message itself is not important to the messaging process. It could be a request to have a method executed, or it could be a document, such as a financial report or a script representing a computer graphics procedure.

Web Service Model

In order to request a service over the Internet, a person must go through a standard procedure. We covered this earlier. Figure 1 suggests the structure of Web Services architecture, which is another standard operational model. There are three roles: the service provider, the service requester, and the service registry. The *service provider* makes a service available over the Internet. The *service requester* consumes a service by sending an XML message to the service provider over the Internet. The *service registry* is a centralized repository of information about services that are available, serving as a computer-oriented version of the traditional “yellow pages.”

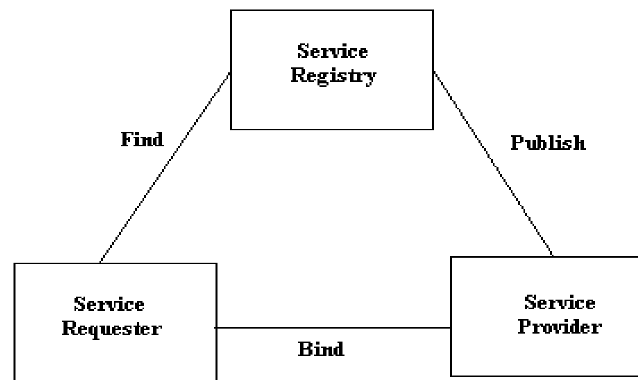

Figure 1. Structure of the Web Service Model.

Associated with the three roles are three activities. The service provider publishes available services in the service registry. The service requester finds out about services by accessing the service registry. The service requester invokes a service (called *bind*) by sending an XML message, referred to above, to the service provider. A familiar example of publish, find, and bind is an online book service. The prospective buyer consults the company's online catalog to find a suitable book. The buyer then finalizes the purchasing process by providing the requisite information, and the seller handles the billing and the physical transportation of the item purchased.

There is more to it, of course. When a service provider publishes service information, it must be described in a form that the service requester can understand. XML is used for this task by employing a document structure known a UDDI, which stands for *Universal Description, Discovery, and Integration.* When a service requester invokes a service, XML is again used in a form of descriptive language known as WSDL, which stand for *Web Services Description Language.*

Web Service Goal

The goal of Web Services was and still is the notion of having computers talk to each other to arrange for a service without human intervention. At this stage, the Internet community has done a commendable job of establishing the requisite technical infrastructure, but the process still requires client interaction at the service-requester end. The focus is currently on building a service-oriented architecture to support future developments.

SERVICE ARCHITECTURE CONCEPTS

Service architecture is a collection of design patterns for constructing services from building blocks that can be shared between service systems. Most business processes already incorporate a form of service architecture, since the principles are derived from ordinary common sense. For example, many accounting departments include component services, such as credit checking and invoicing. When the objective is to align information services with business processes, however, the design gets more complicated and has given rise to a field of study known as *service-oriented architecture* (SOA). The basic idea behind service architecture is that you have a collection of components, representing business functions or computer applications, and you want to fit them together to make a business process or an information system. Components encapsulate services so that a service-oriented application or a business process is assimilated from multiple components that achieve the desired functionality by collectively orchestrating the operation of the needed services. The guiding principle behind service-oriented architecture is that once a component is established, it can be reused in other applications or business processes. Eventually, an organization runs out of components to build so that the synthesis of an application or a business process becomes a matter of piecing the components together – much like the manner in which an aircraft manufacturer or automobile company assembles relatively complicated products from off-the-shelf or specially-designed components. There are two aspects to the idea of building functionality with components; the first is putting the components together, and

the second is making the inherent services interact in such a way that a desired state of business process engineering (BPE) is achieved.

Solution Life Cycle

An effective solution sequence for any development project incorporates a set of well-defined steps, such as the following: requirements analysis, modeling, architectural design, detailed design, construction, and testing. In the modern view of development, incorporating service architecture principles, these steps are divided into two phases: the *preproduction phase*, wherein a set of packaged components are collected, and the *production phase*, consisting of assembly and deployment.

It is important to recognize that the term "production" in the context of service life cycle refers to the synthesis of a business process or the development of an information system, and not to the actual utilization of the process or system, as in everyday operations. So, in a sense, we are producing a solution and not using a solution. Once we have the wherewithal to assemble a solution from components and not have to develop those components from scratch, then we can spend our resources making sure that the eventual solution to whatever problem we are dealing with actually satisfies business needs – and is developed in a reasonable time frame. This is where the term *agility* comes from. The management of an enterprise, for example, perceives that it needs an IT solution to an e-commerce opportunity, and the IT department can expeditiously deliver that solution.

On Demand

The term "on demand" seems to have navigated its way into the business literature in at least three ways. In the first instance, on demand refers to the access of information, such as from the World Wide Web or any other information repository, from wherever the end user may be and whenever the interaction takes place. In the second instance, on demand refers to access to computer application programs without specifically having to purchase them. Also known as *utility computing*, this form of on demand would allow end user to pay only for the use of software, rather than having to purchase it, as is typically the case with traditional office software. Finally, the third instance of on demand and the one in which we are interested refers to the techniques for the rapid development of business processes and computer information systems to support enterprise services.

The flexibility inherent in on demand services provides a payback for most enterprises that is greater than the value of the processes and applications for which the services were originally intended. Overall, on demand processes developed through service orientation can deliver innovation, flexibility, shorter time to market, and also serves as a vehicle for rethinking industry structures.[7] In fact, the business value of service architecture is perhaps best summarized by the following quotation from Cherbakov, Galambos, Harishankar, Kalyana, and Rackham:[8]

What is described here is a business that is able to recognize change as it is occurring and react appropriately, ahead of the competition, and keep pace with demands of its customers, value-net partners, and employees alike. In trying to achieve this state, the business will need to leverage technology to the fullest. We call such a business an "on demand business." Fundamentally, becoming an on demand business is equivalent to achieving total business flexibility. Two important enablers contribute to the realization by an enterprise of this vision of on demand – componentization and service orientation.

Components are related to functions – or to be more specific, business components are related to business functions. In a real sense, therefore, service architecture refers to the deconstruction into components of an existing business system and subsequently its reconstruction into an operational network of cooperating and integrated elements needed for synthesizing responsive enterprise-wide systems.

[7] See Cherbakov, et al. (2005).
[8] Op. cit. p. 654.

Components, Services, and Functions

It's all relatively straightforward: most components encapsulate one or more services; many complex services require more than one component; enterprise processes are constructed from components; and enterprise functions are an amalgamation of corresponding services. The notion of putting components together to achieve some enterprise function is called *composability*, and in order to do so, the methodology demands severe constraints on the manner in which the components are constructed and packaged for reuse. Components must fit together in order to operate as intended; this requirement is known as *interoperability*.

Service Orientation

Many people are going to say that dealing with a collection of interacting components is just going to increase the complexity of their everyday life. After all, they say, why not buy an application program or adopt an established business process and be done with it? On the other hand, there is something to be said for building systems out of packaged components. If a component fails, replace the entire assembly and let the customer – or should be say client – pay for it. After all, that perspective has some merit with products and is widely adopted. The point to remember is, "What's good for products is not necessarily good for services." Here are some of the reasons.

Because services are heterogeneous and involve client interaction, most service interactions are essentially different, so that the unrestricted use of packaged facilities does not automatically contribute to efficiency. With both products and services, features sell packaged facilities, so that if you obtain two related packages, there would normally be a duplication of functionality. In other words, organizations that produce packages, in the most general sense, include as many features as possible to optimize marketability. Most of us THINK products and DO services. Moreover, there is no guarantee that similar components in different packages operate – or interoperate – in exactly the same manner.

Another consideration is that in the area of professional, scientific, and technical services, the operant process is to construct a flexible system, perhaps for a client, in which components can be replaced on a demand basis to satisfy business conditions. In this instance, one would want each component to be designed as granular as possible with a well-defined interface.

SERVICE DEVELOPMENT

One of the basic tenets of service science is that service providers can participate in a service experience by applying knowledge, skill, ingenuity, and experience, without having to invest in the usual encumbrances of product development. In this section, we are going to cover *service development*, without having to necessarily develop each and every service resource. The subject matter primarily concerns "legacy systems," and the methodology applies to just about any kind of service an ordinary person can imagine.

Legacy Systems

Many, if not most, information systems used in business, education, and government are known as *legacy systems* and continue to be the core of enterprise technology. For example, the "grunt" work underlying heavy duty data processing is performed behind the scenes, often during the wee hours of the night by mainframe computers. Linking these systems to modern Web services has been difficult, because they are difficult to change without running the risk of upsetting the applecart of good performance. The programs are written primarily in the COBOL programming language and precise specifications are not always available, adding another dimension to the problem.

Information systems that are cumbersome to change are referred to as being "brittle" and limit an enterprise's ability to respond to changing business requirements. On the other hand, legacy systems are serious assets to an organization and typically represent considerable investments. In many cases, organizations achieve a

level of competitive advantage through the use of legacy systems. Legacy systems support day-to-day operations and incorporate the business logic inherent in all areas of the business model.

Service architecture purports to leverage legacy systems by unlocking the business functionality through loosely-coupled but well-structured service components abducted from legacy systems. The service components can then be choreographed to adapt or extend business processes to satisfy current needs. This can be achieved in two ways: leveraging or repurposing. With *leveraging*, the functions in legacy systems are exposed without rewriting the system. With *repurposing*, the programs are rewritten for the modern world with a modern language, such as Java, for use on servers designed for the Internet and the World Wide Web. Clearly, leveraging is the way to go with legacy systems, because of the risk involved with rewriting large programs and getting it right the first time.

Exposing Functionality in Legacy System

Exposing business services by leveraging legacy systems is not a simple matter and it requires good strategic planning, time, and considerable resources. Typically, the work is outsourced to IT consulting companies, because a high level of expertise is required, but not otherwise needed to sustain enterprise operations.

Although the task is exceedingly complex, the idea is relatively simple: put a software wrapper around the legacy code and expose what you want to expose through well-defined interfaces. The conceptual software wrapper is known as an *adapter.*

Here's how it works. One component needs the services of another component, which may reside locally or be available over the Internet residing somewhere out in cyberspace. The needy component sends an XML message to the servicing component requesting a service of some kind. What this means is that the serving component does something and returns the result as an XML message to the requestor. The messages adhere to an agreed-upon format so that the programs can understand each other.

The overall process is not much different from when one person asks another person for the time. The requestor issues the request in a socially agreed upon language in a well-defined format. The responder accesses a time resource and returns the time in the same language and in a related but different format. If another person asks for the time, the process is repeated.

There are some hidden components in the messaging scenario. The requestor needs information on who would know the actual time. So an internal registry of "people who know about time" is implicitly consulted before the initial message is sent by the requester to the responder. With service architecture, a registry of components is needed to know which components to call upon when a particular service is needed. Part of the registry process could very well involve a search process to determine which registry contains the needed information. Then, perhaps, the requestor might engage a local registry to store pertinent information from non-local registries to facilitate subsequent operations.

SERVICE REFERENCE ARCHITECTURE

A certain amount of structure among components is required for the capabilities, mentioned above, to function together as a coherent whole. It is commonly known as the *SOA Reference Architecture*. The reference architecture is essentially a stack of functionality, implying that service messages flow upward and downward in the stack. [9]

Loose Coupling

The basic principle of service architecture is that synthesis involves composition. A business process or a computer application is created by combining independent components that are loosely coupled. *Loose coupling*, in

[9] The structure of the generic reference architecture is adapted from the webMethods report (see references, p. 5).

this instance, simply means that components – that is, the components providing the services – pass requests and data, in the form of messages, between each other in a standard manner without the need for underlying assumptions that would compromise component operational interdependence. Thus, a small change in the functioning of one component would not require a change to other components that rely on the changed component. With component architecture of this sort, it is important to recognize that components normally relate to enterprise-level processes spanning people, systems, and information.

Services

Services can be created or exposed. In the former case, an organization creates the business process or a computer application from scratch. In the latter case, an existing service is insulated with a logical container and its functionality is explicitly described so that other entities can use it. It's entirely possible that a service developed for another generation doesn't exactly fit in with what you are doing. In this instance, an adapter is needed to make whatever adjustments are necessary to use the service. It's like using your spouse as an adapter to your mother-in-law to arrange for babysitting service. In the world of computers, an *adapter* is a software module that permits access to a service through a standard messaging interface, usually created through the XML language.

Combining diverse services and exposing them as a single service is commonplace in everyday life. Consider, for example, a delivery service that combines three capabilities: dispatcher, driver, and accountant. The dispatcher interacts with the customers and makes the arrangements for the deliveries. The driver organizes his or her delivery route and makes the deliveries. The accountant records transactions, sends bills, and records payments. Yet, to the customer, there is only one service interface, which, in fact, is the point where the package is submitted for transportation. The delivery service has been composed from the three component services and the total process is call *composition.* Facilities, sometimes called *tools*, are needed to put references to components and corresponding services in the registry so that system designers can find them. All of this points to why the subject of web services is so important. Web services with its associated XML, SOAP, UDDI, and WSDL facilities, provide a convenient means of establishing a reference architecture.

Messaging

The messaging layer of the service reference architecture provides the means for the components to interact and emphasizes the need for in-between functionality to provide the requisite level of independence required by SOA. Consider, for example, an investment firm that supports a database of up-to-the-second stock prices. An investment advisor with a client on the line would like the latest price of AT&T stock. So he or she enters the stock exchange code for AT&T, namely 'T', and presses a key on the advisor's workstation. In a flash, the current stock price is returned. Some in-between hardware and software, known as *middleware*, is required to make it all happen. Clearly, many other services within the investment firm would also utilize the same stock price service. It's a simple example but gives evidence that the component-based approach to system development has some definite merit.

A messaging service would normally use *asynchronous messaging*, which means that the requestor sends the message to the service and the result is returned as soon as possible. While the person may be waiting for a response operating in a synchronous mode, the underlying hardware and software goes along its merry way sending messages back and forth asynchronously. Because of the great difference in processing speeds of humans and computers, it appears as though the computer is sitting there waiting for a request and responds immediately. In reality, that request may be put on a queue and processed in order of arrival, recognizing that different methods for queue management may be used.

Registry

The concept of a registry was introduced previously in the context of legacy systems and web services. With web services, the registry is a general facility for storing service information that can be retrieved through XML messaging. It's more complicated but that's the idea. With service architecture, as in the present context, the registry is a repository for information on components, intended for persons synthesizing a composite service, and

additionally contains tools to assist in achieving that synthesis. *The registry is a data base of components and the services they supply.*

The registry should also contain facilities for convenient search, the import and export of entries, and change management. In the latter instance, it is necessary for users to be informed of updates that might affect their performance.

The registry should additionally reflect business policy as it refers to distribution, security, and ownership.

Architecture Services Management

In the present context, services management refers to the operation and management control over business processes constructed with a service orientation. The efficacy of service operations is always of concern as it relates to service-level agreements as they relate to performance and quality of service. In the former case, performance encompasses the availability and reliability of individual services. In the latter case, quality of service refers to the statistical analysis of specific service events.

Management control reflects governance concepts as they apply to the operations mission, previously mentioned. Governance should reflect the fact that service architecture is a methodology for using services to construct services and has two major focuses: (1) The creation of processes, operating in the form of services, to support both IT-enabled and non-IT-enabled business activities; and (2) The control and support of the business services through a formal process for managing services. To summarize, governance provides support for empowering people to do what they do in line with organizational objectives.

Orchestration

Orchestration refers to the dynamic linking of services together to achieve a business purpose. The business processes are layered on top of the services, so in a sense, the services are anchored into the processes. In IT-enabled processes, the business process is a script written in a "business process execution language" that successively calls the needed components in order to invoke the services constituting the business function. This combinatory operation was earlier referred to as composition and can be conceptualized as the workflow of services. In a non-IT enabled process, the composition is achieved through management directed policies, procedures, and business rules.

Actually, the term "orchestration" has two meanings in the context of service science. Let's take a computer application as an example. The first meaning has to do with setting up the structure as a controlling module that successively invokes services to achieve a business objective. The components do not have some form of inherent stickiness that enables operational affinity among loosely-coupled components in a meaningful order. That is where the business process execution language (BPEL), referred to just above, comes in. The application designer has to set up the service chain beforehand.

This is where the operational structure in Figure 7.1 comes in. The service bus effectively connects the registry, workflow, composition, and the underlying system (called the *platform*) as pieces that do the work to construct function from components viewed as services. The second meaning of the term "orchestration" refers to the actual running of the application. The BPEL script is actually executed by an operational entity, intermediate data is stored in an operational database (not shown), and the business result is achieved.

Analysis

One of the facets of the service domain is that service quality is directly related to client interaction and involvement. This requires constant tweaking, otherwise known as continuous improvement (i.e., *kaizen* in operations management). Business performance is constantly monitored – there is nothing new about this. With service systems, however, the raw operational data is frequently embedded deep down in independently constructed loosely-coupled components. Getting this data out for analysis is a task that should be addressed at the design level.

User Interaction

On the surface, the end-user interface development model seems simple. All that needs to be done is to construct a prototype, test it, improve it, and then have the end-user group sign off on it. With service-oriented architecture, however, the *user interaction* is with a business process, which is a notch up from what normally is construed as the end-user interface. The term "user," therefore, refers to the user of a service, and not necessarily to the user of an application. As in the restaurant example, the user of a service doesn't have to be the customer.

The user of a service can be another service, which leads to the notion of a service architecture in which components can be assembled without the use of special adapters.

SERVICE ARCHITECTURE PRINCIPLES

The use of design principles is paramount to the construction of a successful service project. Otherwise, service systems development is another "random walk down Wall Street." Here is a set of service architecture principles.

Service Abstraction. The key benefit of service abstraction is that "inside" information about a component is effectively hidden from the outside so that component can be used by other diverse services. This principle is sometimes referred to as information hiding. Often, internal operational details are superfluous to a referencing service where only a result is needed. Take the credit checking service as an example. An outside user of that service is usually only interested in the credit worthiness of a subject and not in the procedures and file processes necessary to ascertain that rating. In fact, operational details may change without the requester knowing or caring about them. The concept of abstraction applies to other organizational functions and computer modules, as well.

Service Encapsulation. Service encapsulation enables a service – often bundled as part of a larger operational entity – to be referenced via an adapter to preserve and take advantage of previously developed functionality. As with the preceding principle, encapsulation may apply to organizational as well as informational components.

Service Loose Coupling. This principle simply demands that components are not implicitly dependent upon one another, such that use by a non-coupled component is prohibited. Another way of expressing the concept of loose coupling is one component does not require that another component be in a particular state at the time of invocation.

Service Contract. The concept of a service contract reflects that it is necessary that a complete specification be made of the precise services provided by a service component and exactly how those services are to be addressed. A service contract describes how two components are to interact. With Web services, the contract refers to a WSDL (Web Services Definition Language) definition and a specification of the XML schema definition of precisely how messages between a requester and the repository are to be formatted.

Service Reusability. Service reusability simply refers to the practice of designing a component so that it can be used in more than one place. In general, the intention is to provide services that can be used by more than one business process.

Service Composability. Service composability refers to the combining of services to form composite services. This practice implicitly imposes a restriction on the component services so that they adhere to the specifications in the service contract. Service composition is usually performed to synthesize a business process.

Service Autonomy. Service autonomy is conceptually modeled after the human nervous system and refers to a component's capability to self-govern its own operational behavior. Autonomy reduces the complexity of business processes composed from self-regulating components. Autonomy allows a business process to provide a higher level of productivity by being able to manage itself. This is a tricky principle, because the implication is that a component just operates on its own as some artificial intelligence robot. For most services, this principle simply means that a service invoked through some form of "service bus" takes its input parameters and performs its functions, as specified in its service contract, without requesting additional input or operating instructions.

Service Discoverability. Service discoverability is a complex arrangement of being describable, via the service contract, and being accessible via a registry and a description language. Essentially, this means that the description of a service, found through a search process, additionally provides information on how to use a service.

SERVICE ARCHITECTURE STRUCTURE AND OPERATION

A business process is composed of one or more business services frequently implemented through information and communications technology (ICT). Krafzig, Banke, and Slama[10] state the modern dependence on ICT in the following way. "… enterprises heavily depend on the IT backbone, which is responsible for running almost all processes of modern enterprises, be they related to manufacturing, distribution, sales, customer management, accounting, or any other type of business process." This section introduces the concept of enterprise systems and then presents definitive information on the structure and operation of service architecture in an enterprise environment.

Enterprise Systems

An enterprise system cuts across the total organization and encompasses inter-departmental dependencies and relationships with suppliers and business partners. Accordingly, the enterprise software should be tightly coupled with the organization, but not with itself, based on the component model. This reflects the agility and incremental change that we referred to earlier. We require a structure that promotes loose coupling through messaging and platform interoperability.

Service Architecture Structure

The key structural elements in a service system are the services, a service repository, the service broker, the service bus, the service manager, and the interface elements. The interface elements can be to end users or to application programs.

From a structural viewpoint, the *service* provides business logic and consists of an implementation and a service contract. The *service repository*, operating as a virtual library, exists as a place to store service information and how to retrieve that information. The service repository certainly has a computer flavor to it, but that need not be the case. Many service firms have manual lists of the services they offer. In the computer version of a service repository, however, the storage facility could be accessed manually during development and dynamically during the execution of a component. The *service broker* connects services together by accessing the service repository for information about services and providing the linkage to connect components. The *service bus* is the nerve center is an enterprise system and is covered separately, as is the service manager, which is the mechanism by which enterprise processes are constructed. The interface elements are the input and output to the system.

The term "interface" normally implies an end-user interface with which most persons are familiar. In the case of enterprise systems, however, an interface can be to another computer application, a database, or a legacy system.

Enterprise Service Bus

An *enterprise service bus* (ESB) is a collection of ICT facilities for routing messages between services, or more specifically between components. The bus metaphor is apt in this case. The message gets on, goes to its destination, and gets off. The metaphor ends there, because there are different kinds of busses and unique things happen on different busses.

The most straightforward kind of service bus is a high-speed data link between services, as alluded to earlier in the stock broker example. The stock broker needs the current price of a stock for an ongoing transaction. The stock symbol is entered into a workstation and a button is pressed. In a fraction of a second, the current price is

[10] See their book in the selected readings, p.1.

returned by a service connected to the other end of the service bus. The service bus in this instance, is a combination of hardware and software often referred to as middleware. In this model of bus, the service bus could also be a specially constructed data link between business partners or between organizational units, termed electronic data interchange (EDI).

The most general form of ESB, however, uses the Internet with all of its inherent requirements for interoperability. In this instance, a message, perhaps requesting a service, may go through a necessary protocol conversion in its route from sender to receiver. Another possible function performed by an ESB is *context mediation*, which refers to a change in value based on contextual differences. An example would be the change of a price from Yen to Dollars during message processing.

Another related topic is *web based intermediary*, or WBI for short. A WBI is a program that runs in concert with a client's browser and acts as a form of software assistant, filtering and preparing information to satisfy particular needs.

Service Manager

The most prevalent use of service architecture is to construct computer applications. The service manger ties everything together and runs the show. Clearly, this is an operational function but a structural component is needed to do it. In a sense, the service manager is the "main program" of an application. The service manager could be a specially written component in an enterprise system, or it could be a vendor-supplied package that successively calls upon required services.

Service Architecture Operation

An enterprise system is sometimes referred to as an "end to end" operation that represents a business process. Another means of conceptualizing an enterprise system is that it is controlled process flow. As covered above, the service manager controls the process flow through a process called *orchestration*. The conductor of an orchestra controls the activity of a set of musicians through minute actions termed orchestration. The same concept can be applied to the execution of an enterprise system.

Orchestration is different than choreography. Choreography refers to what a collection of services can do, and orchestration refers to precisely when and how they actually they do it.

A business process can be scripted in a language, such as BPEL, or written in a computer programming language. Business Process Execution Language (BPEL) is an XML-based scripting language for orchestrating service applications.[11]

QUICK SUMMARY

The purpose of this overview is to present a bird's eye view of service technology and architecture. The principles inherent in this viewpoint are summarized in the following quick summary.

1. Services are ubiquitous but require messages to communicate information between client and provider. A client and a provider can be tightly coupled, as when a patient is sitting in front of the doctor and they are having a give-and-take conversation, or loosely coupled, as when you send a request to someone via email and receive a response at some undetermined time in the future. In the former case, the client and provider are communicating in a *synchronous mode* without technology, and in the latter case, they are communicating in an *asynchronous mode* with the use of technology.
2. The focus is on the data that is transmitted and not on the communications medium, which can take the form of a human interaction or a computer-based message. The context for the message can be embedded in the message or inherent in the way that the service provider is addressed. It is useful to recognize that

[11] For a good reference to BPEL, see Margolis (2007).

we are operating at two levels: the service level and the message level. At the *service level*, the message entity that receives the message is the service provider, and in the case of a computer, it is regarded simply as the service. At the *message level*, there is some choreography involved with providing a service, as demonstrated by the above two-step interaction. In fact, a service may involve the interchange of several messages.

3. In its most simple form, a message is a string of characters encoded using standardized coding methods commonly employed in computer and information technology. Messages have a uniform format consisting of a header and a body. The *header* primarily concerns addressing and includes the address of the sender and the receiver. In the request/reply message pattern, the return address is picked up from the message header for the response portion of the transaction. The *body* of the message contains the information content of the message, and because it is intended only for the receiver, it is not usually inspected during message transmission.
4. A service that takes place on the Internet and the World Wide Web is called a *web service*. A web service is a process in which the provider and client interact to produce a value; it is a pure service. The only difference between a web service and medical provisioning, for example, is that, in the former case, the client and provider are computer systems. The most pervasive web service computer application on the Internet is electronic mail, commonly known as email. The most widely used application on the World Wide Web is to find information. The major web technology tools and techniques are HTTP, HTML, and XML.
5. *HyperText Transfer Protocol* (HTTP) is a collection of rules and procedures for transferring messages between computers over the World Wide Web. Without HTTP, the web would not be the revolutionary phenomena that it is today.
6. It is possible to send a document from one computer to another and have that document displayed on the receiving end in a reasonable form without regard to the brand and model of computer, kind of software, time of day, and location. This feat is possible because of hypertext markup language (HTML).
7. Extensible Markup Language (XML) is a language and a standard for service messaging. Whereas HTML describes how a document will be rendered on the receiving end of a message, XML gives the semantics (or meaning) of a document.
8. A Web Service is any service that is available over the Internet, uses a standard XML messaging system, and is not dependant upon any one particular operating system.
9. *Service architecture* is a collection of design patterns for constructing services from building blocks that can be shared between service systems. The basic idea behind service architecture is that you have a collection of components, representing business functions or computer applications, and you want to fit them together to make a business process or an information system.
10. Components encapsulate services so that a service-oriented application or a business process is assimilated from multiple components that achieve the desired functionality by collectively orchestrating the operation of the needed services. The guiding principle behind service-oriented architecture is that once a component is established, it can be reused in other applications or business processes. Eventually, an organization runs out of components to build so that the synthesis of an application or a business process becomes a matter of piecing the components together.
11. The term "on demand" seems to have navigated its way into the business literature in at least three ways. In the first instance, on demand refers to the access of information, such as from the World Wide Web or any other information repository, from wherever the end user may be and whenever the interaction takes place. In the second instance, on demand refers to access to computer application programs without specifically having to purchase them. Also known as *utility computing*, this form of on demand would allow an end user to pay only for the use of software, rather than having to purchase it, as is typically the case with traditional office software. Finally, the third instance of on demand and the one in which we are interested refers to the techniques for the rapid development of business processes and computer information systems to support enterprise services.
12. It's all relatively straightforward: most components encapsulate one or more services; many complex services require more than one component; enterprise processes are constructed from components; and enterprise functions are an amalgamation of corresponding services. The notion of putting components together to achieve some enterprise function is called *composability*, and in order to do so, the methodology demands severe constraints on the manner in which the components are constructed and packaged for

reuse. Components must fit together in order to operate as intended; this requirement is known as *interoperability*.

13. An enterprise is service oriented if it can be properly viewed as a set of services connected to produce a specific result. Similarly, a computer application or information system is service oriented if is constructed from interacting components running on the same platform or accessible from different platforms via networking facilities.
14. Service architecture purports to leverage legacy systems by unlocking the business functionality through loosely-coupled but well-structured service components abducted from legacy systems. The service components can then be choreographed to adapt or extend business processes to satisfy current needs. This can be achieved in two ways: leveraging or repurposing. With *leveraging*, the functions in legacy systems are exposed without rewriting the system. With *repurposing*, the programs are rewritten for the modern world with a modern language, such as Java, for use on servers designed for the Internet and the World Wide Web. Clearly, leveraging is the way to go with legacy systems, because of the risk involved with rewriting large programs and getting it right the first time.
15. A certain amount of structure among components is required for the capabilities, mentioned above, to function together as a coherent whole. It is commonly known as the *SOA Reference Architecture*.
16. The use of design principles is paramount to the construction of a successful service project. A set of service architecture principles includes the following elements: service abstraction, service encapsulation, service loose coupling, service contract, service reusability, service composability, service autonomy, and service discoverability.
17. An enterprise system cuts across the total organization and encompasses inter-departmental dependencies and relationships with suppliers and business partners. Accordingly, the enterprise software should be tightly coupled with the organization, but not with itself, based on the component model. The key structural elements in a service system are the services, a service repository, the service broker, the service bus, the service manager, and the interface elements. The interface elements can be to end users or to application programs. An *enterprise service bus* (ESB) is a collection of ICT facilities for routing messages between services, or more specifically between components.
18. An enterprise system is sometimes referred to as an "end to end" operation that represents a business process. As covered above, the service manager controls the process flow through a process called *orchestration*. Orchestration is different than choreography. Choreography refers to what a collection of services can do, and orchestration refers to precisely when and how they actually they do it.

ACKNOWLEDGMENT

Thanks to William Hahn and Margaret Katzan for reading the manuscript.

REFERENCES

1. Carter, S., The *New Language of Business*, Upper Saddle River, NJ: IBM Press, 2007.
2. Cerami, E., *Web Services Essentials*, Sebastopol, CA: O'Reilly Media, Inc., 2002.
3. Cherbakov, L., Galambos, G., Harishankar, R., Kalyana, S., and G. Rackham, "Impact of service orientation at the business level," *IBM Systems Journal*, Vol. 44, No. 4, 2005, pp. 653-668.
4. Dykes, L. and E. Tittel, *XML for Dummies* (4th Edition), Hoboken, NJ: Wiley Publishing, Inc. 2005.
5. Erl, T., *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, Upper Saddle River, NJ: Prentice Hall, 2004.
6. Erl, T., *SOA: Principles of Service Design*, Upper Saddle River, NJ: Prentice Hall, 2008.
7. Ernest, M. and J.M. Nisavic, "Adding value to the IT organization with the Component Business Model, *IBM Systems Journal*, Vol. 46, No. 3, 2007, provider.387-403.
8. Gottschalk, K., Graham, S., Kreger, H., and J. Snell, "Introduction to Web services architecture," *IBM Systems Journal* (Vol. 41, No. 2), 2002, pp 170-177.
9. Hagel, J. and J.S. Brown, *The Only Sustainable Edge*, Boston: Harvard Business School Press, 2007.
10. Hurwitz, J., Bloor, R., Baroudi, C., and M. Kaufman, *Service Oriented Architecture for Dummies*, Hoboken, NJ: Wiley Publishing, Inc., 2007.

11. IBM Corporation, *Extend the value of your core business systems: Transforming legacy applications into an SOA framework*, Form G507-1950-00, September 2006.
12. Krafzig, D., Banke, K., and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices*, Upper Saddle River, NJ: Prentice Hall, 2005.
13. Margolis, B. with J. Sharpe, *SOA for the Business Developer: Concepts, BPEL, and SCA*, Lewisville, TX: 2007.
14. McGrath, M., *XML in Easy Steps*, New York: Barnes & Noble Books, 2003.
15. Musciano, C. and B. Kennedy, *HTML: The Definitive Guide*, Sebastopol, CA: O'Reilly Media, Inc., 1998.
16. Potts, S. and M. Kopack, *Web Services in 24 Hours*, Indianapolis: Sams Publishing, 2003.
17. Smith, J., *Inside Windows Communication Foundation*, Redmond, WA: Microsoft Press, 2007.
18. Spohrer, J., *Service Science, Management, and Engineering (SSME): State of the Art – service science*, IBM Nordic Service Science Summit, Helsinki, Finland, February 28, 2007.
19. Van Slyke, C. and F. Bélanger, *E-Business Technologies: Supporting the Net-Enhanced Organization*, New York: John Wiley and Sons, Inc., 2003.
20. Watt, A., *Teach Yourself XML in 10 Minutes*, Indianapolis: Sams Publishing, 2003.
21. webMethods, *SOA Reference Architecture: Defining the Key Elements of a Successful SOA Technology Framework,* www.webMethods.com, 2006.
22. Wikipedia, *HTTP*, www.wikipedia.org, 2007.
23. Woods, D. and T. Mattern, *Enterprise SOA: Designing IT for Business Innovation*, Sebastopol, CA: O'Reilly Media Inc., 2006.

NOTES

NOTES