In the first article in this series, you learned about how current web services infrastructure can be used to support asynchronous behavior. This time around, Holt Adams provides concrete blueprints that will help you build your own asynchronous web services. These practical patterns can be used today for handling responses to web service requests as separate transactions. You'll also learn which real-world situations each pattern lends itself to.

In the first article in this series, I discussed the nature of asynchronous operations and their application to web services (see Related topics). In some situations, responses to web service requests are not provided immediately, but rather sometime after the initial request transactions complete. Such asynchronous operations aren't explicitly supported by web services specifications and standards; however, those standards do include the infrastructure and mechanisms on which asynchronous operations can be based. The first part of this series should have given you an idea of how you could use that existing infrastructure to support asynchronous behavior; if you haven't read it yet, I urge you to do so, as that information will help you follow this current installment.

In this paper, I'll examine several design patterns for asynchronous web services. These patterns will help developers of client and service provider applications to architect their code to support asynchronous behavior while taking into consideration code complexity, use of given transports, and the need for explicit acknowledgments. You may also be interested in reading about another package that supports asynchronous behavior in web services as explained in sidebar 1.

Web Service asynchronous patterns and transmission primitives

The four patterns for support of asynchronous web service operations discussed here are based on the four transmission primitives that an endpoint can support, as defined in version 1.1 of the Web Services Descriptor Language (WSDL) specification (see the Related topics section below for a link to this spec):

- **One-way**: The endpoint receives a message.
- **Request/response**: The endpoint receives a message, and sends a correlated message.
- **Solicit/response**: The endpoint sends a message, and receives a correlated message.
- **Notification**: The endpoint sends a message.
IBM’s Web Service Invocation Framework and asynchronous operations

The Web Service Invocation Framework (WSIF) provides a client API for invoking web services asynchronously using a local proxy (for example, a stub) with the capability to use different transports based on the context of the invocation. Many of the patterns addressed in this paper can be supported by WSIF components, thus allowing true support for asynchronous web services operations.

For more information on WSIF, see the Related topics below.

It should be noted that the number of transmission primitives and the number of patterns discussed in this paper are totally independent of each other.

Each of the patterns introduces a correlator exchanged between the client and service provider for use in associating responses with requests. The correlator can be provided by either end of the exchange and its creator may be determined based on the underlying transport. For example, when using HTTPR and JMS, the source of a message provides the correlator: a transaction ID or a combination of JMSMessageID and JMSCorrelationID.

For single-direction operations, if you are using HTTP or HTTPS and the reception of the service invocation needs to be confirmed by the client, the client's HTTP protocol handler should block on the invocation waiting for the HTTP response (for example, a 200 status code) to ensure that the request has been successfully received by the service provider's HTTP listener. For a case in which a service proxy is used by the client, any error conditions associated with the request should result in an exception being thrown. It’s important to model the interface of the service proxy to match the WSDL operations defined by the service provider. For example, if a client invokes a one-way operation, the proxy should never return a parameter (for example, a status code) to the client. Such an exchange would in effect make the operation a request/reply operation, with the reply information not coming from the service provider.

The W3C's WSDL working group is expected to expand the language's support for asynchronous operations by providing the ability to define callback mechanisms (for example, reply-to addresses) formally within WSDL. In the meantime, the four primitives listed above can be used in support of asynchronous operations. However, the IDEs and other web services tooling currently available to automate the generation of client-side service proxies typically only support the request/response model.

Pattern 1: One-way and notification operations

In this pattern, the request and the response are two messages defined within separate WSDL operations. The request is modeled as an inbound one-way operation and the response is modeled as an outbound notification operation. Each message is sent as a separate transport-level transmission.

This pattern (see Figure 1) provides a high level of decoupling between the client and service provider, as it supports the use of two datagrams exchanged between the parties, one for the request and one for the response.
For this pattern, the client is responsible for creating the correlation ID and passing it to the service provider via whatever mechanism has been agreed upon by the two parties. SOAP headers, HTTP headers, and JMSCorrelationIDs would all be acceptable mechanisms.

Defining the reply-to address, which indicates where the response should be sent, is also the responsibility of the client, and the means for informing the service provider of this address is determined by how the WSDL is defined for the operations. If the client has published a notification listener service supporting a one-way operation, its WSDL will contain the port address for the service. Likewise, the service provider will need access to the WSDL of the client's service to determine where to send the response. Access to the WSDL of the notification listener service can be provided when the provider's web service is deployed or at runtime by passing a reference to the WSDL on the initial request. Alternatively, the specific address (for example, the URI) denoting where the response is to be sent can also be provided explicitly as a parameter on the request.

This pattern is also applicable for publish and subscribe (pub/sub) and event notification types of services. A market index update application would be a good example of a pub/sub service; examples of event notification services include applications that notified interested parties about the completion of or exceptions in business process tasks, the completion of a long-running report, or the meeting of certain inventory thresholds. Providing the reply-to address information as a parameter on the request (a request to subscribe to a topic or event, for instance) will enable a service provider to support a large number of subscribers with little administrative support.

For this pattern, in which messages are sent using separate transport-level transmissions without application-level acknowledgements being exchanged between the client and service provider, the transport used should be one considered reliable if the business process that's being supported by the message flows is critical.

**Pattern 2: Request/reply operations**

In this pattern, request and response are two messages defined within a single request/reply operation and sent as two separate and unrelated transport-level transmissions.
This pattern (see Figure 2) can also provide a high level of decoupling between the client and service provider, as it supports the use of two datagrams exchanged between the parties for the request and response. However, to use this pattern, the service provider must be a little more sophisticated in processing information at runtime. For example, the service provider will need to be able to handle as an input parameter the address to which it should send the response (for example, the reply-to address).

**Figure 2. Request/reply operation**

For this pattern, the client is responsible for creating the correlation ID and passing it to the service provider via whatever mechanism has been agreed upon by the two parties. Again, SOAP headers, HTTP headers, and JMSCorrelationIDs are among the acceptable mechanisms.

Defining the reply-to address denoting where the response should be sent is also the responsibility of the client. Since a single operation is used for this pattern, a reference to the address or the explicit address itself must be provided as a parameter on the request. For example, if the client has a published an asynchronous response listener service supporting one-way operations, a reference to the service’s WSDL can be provided on the initial request.

This pattern is applicable for general-purpose services where the request results in a single response; examples of such services include the persistence or retrieval of data, or the initiation of a business process consisting of a single unit of work, such as an electronic payment. Pattern 2 is similar to Pattern 1, in which messages are sent using separate transport-level transmissions without application-level acknowledgements being exchanged between the client and service provider. Thus, the transport used for this pattern should also be one considered reliable if the business process that’s being supported by the message flows is critical.

**Pattern 3: Request/reply operations with polling**

In this pattern, request and response are handled using four messages defined within two separate WSDL operations. The initial request is modeled as a request/reply operation, with two messages (a transmission with a reply) sent as a single transport-level exchange. The response is retrieved by a second request, also modeled as a request/reply operation with two messages sent as a single transport-level exchange. The two operations are meant to be implemented as synchronous flows, with information being returned from the service provider for each request providing the client with a level of acknowledgement per request.
This pattern (see Figure 3) enables the client-side implementation to be simpler in support of self-service based solutions, in which the client application initiates all interactions, while also providing a level of decoupling between the client and service provider. However, it's assumed that the request/reply operations are synchronous such that the flows for the reply messages use the native transport reply mechanism (for example, an HTTP Response operation).

**Figure 3. Request/reply operation with polling**

In the example in Figure 3, the service provider generates the correlation ID and the client is responsible for using it to retrieve the response; however, theoretically either side of the exchange could create the correlation ID.

This pattern results in a simpler client implementation, as the notification mechanism and listener components are not required. However, the client must implement a facility with which it can periodically poll for the response from the service provider. This pattern may not be the most efficient, as more than one request per response may be needed to retrieve the response if the initial service request hasn't completed, but it makes for simpler implementations. Thus, this pattern is applicable in cases where simplicity has a priority and where the expected load for the service is low. Examples of service types that could benefit from polling include the initiation of long-running business processes, requests for generation of complex reports, and services used by browser-based customer facing solutions.

**Pattern 4: Request/reply operations with posting**

Under this pattern, request and response are handled using four messages defined within two separate WSDL operations. The initial request is modeled as a request/reply operation, with two messages sent as a single transport-level exchange. The response is modeled as a solicit/reply operation, with two messages sent as a single transport-level exchange. The two operations are meant to be implemented as synchronous flows with information being returned from the consuming party for each request providing the requesting party with a level of acknowledgement per request.

This pattern (see Figure 4) is similar to Pattern 1 and useful for pub/sub or event notification services when a synchronous transport is used and when the client and service provider require
an application-level acknowledgement. Because of this similarity, the example situations given under Pattern 1 could also be addressed by Pattern 4; other examples of service types requiring explicit acknowledgements include services used to exchange business-critical information or confidential information for medical or financial industries -- funds transfers or the initiation of insurance claims, for example.

**Figure 4. Request/reply operations with posting**

![Image of request/reply operations with posting]

If the roles of the client and service provider are reversed for the handling of the response, the WSDL operation for sending the response can be defined as a request/reply operation for the client. The role reversal is simply for convenience; it facilitates the development of the pattern, as today’s tooling does not support solicit/reply operations. The messages that flow between the two parties are not changed; only this article's model for describing the client's and service provider's perspectives is different.

**Conclusions**

The support of asynchronous web service operations can be implemented using both synchronous and asynchronous transport protocols. The use of asynchronous transports, which inherently provide the correlation of request and response messages and provide the mechanisms to query status and retrieve response messages independently, makes the support of asynchronous operations on both the client and service provider sides easier, as message-oriented middleware provides reliable messaging for the transport of web service requests and responses. Likewise, synchronous transports can be used to support simpler implementations that
Asynchronous operations and web services, especially where a self-service style is preferred by the client application.

The table below summarizes the four patterns addressed in this paper, with comments and considerations for solution providers.

<table>
<thead>
<tr>
<th>Asynchronous Pattern</th>
<th>Applicable Transports</th>
<th>Usage Examples</th>
<th>Comments and Considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. One-way and notification operations</strong></td>
<td>HTTPR, JMS, MQSeries Messaging, HTTP, HTTPS, RMI/IIOP, SMTP</td>
<td>Pub/sub; event notification</td>
<td>Inherently supported by asynchronous transports. When delivery of a request or response does not need to be guaranteed, synchronous transports can be used.</td>
</tr>
<tr>
<td><strong>2. Request/reply operations</strong></td>
<td>HTTPR, JMS, MQSeries Messaging, HTTP, HTTPS, RMI/IIOP, SMTP</td>
<td>General service with a single response; transactional services</td>
<td>Inherently supported by asynchronous transports. When delivery of a request or response does not need to be guaranteed, synchronous transports can be used.</td>
</tr>
<tr>
<td><strong>3. Request/reply operations with polling</strong></td>
<td>HTTP, HTTPS*, RMI/IIOP**</td>
<td>Self-service</td>
<td>The simplest client-side implementations. Application-level acknowledgements easily supported with synchronous transports (for example, by HTTP response).</td>
</tr>
<tr>
<td><strong>4. Request/reply operations with posting</strong></td>
<td>HTTP, HTTPS*, RMI/IIOP**</td>
<td>Pub/sub; event notification</td>
<td>Simpler client-side implementations. Similar to Pattern 1, but with explicit application-level acknowledgements for use with synchronous transports.</td>
</tr>
</tbody>
</table>

*: Supported by current web services tooling.

**: Though you can also use asynchronous transports for these patterns, it would be impractical to do so, since the asynchronous transports provide mechanisms that make Patterns 1 and 2 easier to implement.

As noted in Part 1 of this series, future versions of web services standards may simplify support for asynchronous transports (see Related topics below). But with the patterns outlined in this paper, you should be able to start building asynchronous Web patterns for a variety of purposes today.
Related topics

- Read the first part of this article, Asynchronous operations and web services, Part 1.
- You can find a Java Messaging Service API tutorial at Sun Microsystems’ Web site.
- The Java Community Process's site has more information on the Java Connector Architecture.
- For more on programming J2EE APIs with WebSphere, with a particular look at messaging with JMS, WebSphere, and MQSeries, check out this IBM Redbook (in PDF format).
- The W3C has more information on version 1.1 of Web Service Description Language (WSDL).
- Check out IBM's Web Services Invocation Framework, an alphaWorks technology.

© Copyright IBM Corporation 2002
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)