SOAP Nodes in WebSphere Message Broker V6.1, Part 4: Runtime behavior

Rob Henley
Matthew Golby-Kirk

February 11, 2009

SOAP nodes in WebSphere Message Broker V6.1 send and receive SOAP-based Web services messages, enabling a message flow to interact with Web service endpoints. The messages may be plain SOAP, SOAP with Attachments (SwA), or Message Transmission Optimization Mechanism (MTOM). You can configure the nodes using WSDL, and they support the WS-Security and WS-Addressing standards. This four-part series describes the SOAP nodes, the logical tree for the new SOAP domain, configuration, and runtime behavior. Part 4 describes runtime validation, performance, scalability, message flow design, and use of WS-Addressing.

SOAP nodes: Review

In IBM® WebSphere® Message Broker V6.1, the SOAP nodes use the new SOAP parser and SOAP logical tree format. The new nodes are:

- **SOAPInput** and **SOAPReply** -- Used together to provide (implement) a Web service.
- **SOAPRequest** -- Used to invoke a Web service.
- **SOAPAsyncRequest** and **SOAPAsyncResponse** -- Used together to invoke a Web service asynchronously (which simply means that the message flow is not blocked at the SOAPAsyncRequest whilst waiting for the response).

This article builds on the information in Part 1, Part 2, and Part 3 of this series, and discusses the runtime characteristics of message flows using SOAP nodes, including performance, scalability, routing, and error handling.

Validation

At runtime, each Web service message is checked against the WSDL that you used to configure the node. A message that does not correspond to one of the expected WSDL operations causes an exception to be thrown. This checking cannot be turned off.

In addition, the content of the SOAP Body (the payload) and any SOAP header blocks can be checked against the message set that contains the WSDL. This checking is configurable on the

© Copyright IBM Corporation 2009

Validation tab of the node. The key values for the **Validate** option are **Content and Value** (the default, meaning validation is on), and **None** (validation is off). The values **Content** and **Content and Value** are equivalent in the SOAP domain.

For example, consider a SOAP Body with the child `<ns1:payload>`. The validator checks in the message set for a definition of `payload` in the namespace defined by the namespace prefix `ns1`. (This definition will have been added to the message set if necessary when the WSDL definition was imported or generated.) The detailed content of the element is checked and an exception is thrown if the element does not match its definition. (Actually, the validation Failure action is configurable, but the default is to throw an exception.)

The same behavior applies to any SOAP header blocks, or other places in the SOAP message where arbitrary elements are allowed (these are often referred to as extensibility elements). Headers that are not defined in the message set should still be valid, because not all header blocks need be defined in the WSDL. Fortunately this is exactly how extensibility elements are handled: if no definition is found for such an element, then it is assumed to be valid.

There is a small runtime cost if validation is enabled, but the advantages are significant. They include:

- You need less error handling code in your message flow because you know that only valid messages will be propagated.
- The elements in the resulting message tree have their correct XML Schema-defined data types, as opposed to the generic `CHARACTER` data type. (Actually this is only true if you check **Build tree using XML schema data types** on the **Parser Options** tab. Although this is also selected by default, it can be disabled inadvertently by turning validation off and on again.) Use of the correct data types can be significant: for instance, a base64Binary value (perhaps sent in an MTOM document) will be correctly represented as a BLOB.

The Validate setting on a SOAPRequest or SOAPAsyncResponse node applies to the validation of the response message. The validation setting for the outgoing request (SOAPRequest or SOAPAsyncRequest) is inherited from the preceding part of the flow (typically from the selection on an input node or a Compute node).

Performance

Sometimes it is reasonable to disable validation to improve performance. One consequence of disabling validation is that you can then configure one or more **Opaque** elements on the **Parser Options** tab. An opaque element is not parsed at runtime. For example, if your message flow only needs to look at one SOAP header block before passing on the message, you could define other elements such as the SOAP Body content as opaque.

Only pay for what you use

If your message flow does not need validation, security, or WS-Addressing capabilities, then you can improve performance by not selecting these features.

Of course, performance is also improved if you do not select any security features or specify that WS-Addressing is being used. Only select these features if your message flow actually requires them.

As discussed in Part 1: SOAP node basics, there are also performance considerations when deciding whether to use the single SOAPRequest node or the separate SOAPAsyncRequest and SOAPAsyncResponse nodes.

Scalability

Each SOAPInput node is associated with its own thread. If you have six SOAPInput nodes then you will have six threads by default. If you specify six additional instances on the flow, then these additional threads are shared amongst the nodes as required. If you have six SOAPInput nodes and six additional instances on the flow, you will have up to 12 threads. There is a limit of 256 additional instances.

Alternatively, you can specify six additional instances on each node. In this case, these are associated with the individual nodes. If you have six SOAPInput nodes each with six additional instances you will have up to 42 threads.

Each execution group has one listener, and two ports -- one HTTP and one HTTPS. The default SOAP node port numbers are 7800 for HTTP and 7843 for HTTPS. You can change these using the command `mqsichangeproperties YOURBROKER`, followed by:

- `-o BrokerRegistry -n httpConnectorPortRange -v 7800-7842` (for HTTP)
- `-o BrokerRegistry -n httpsConnectorPortRange -v 7843-7884` (for HTTPS)

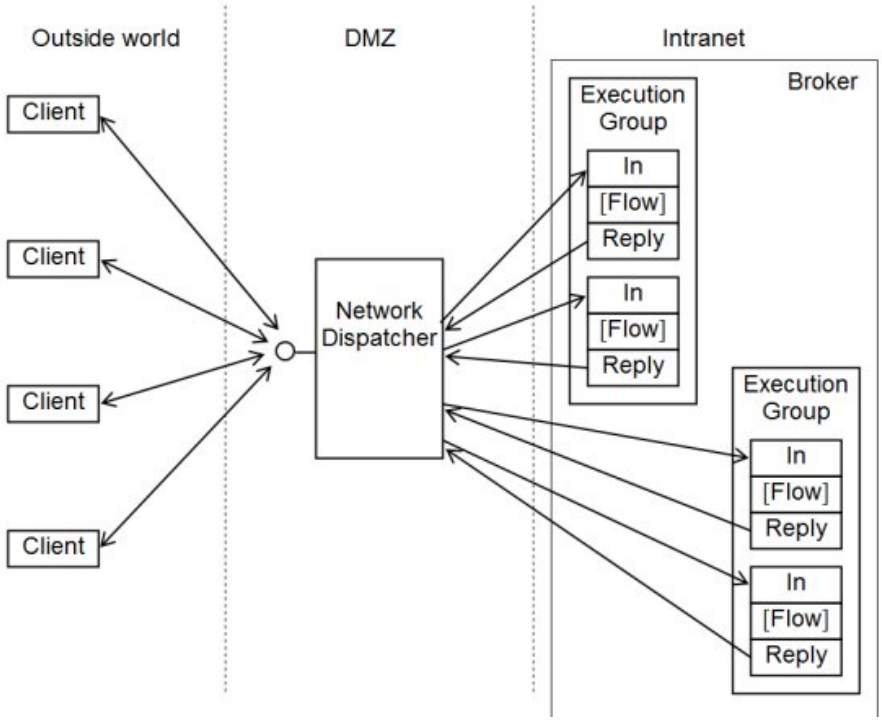
A range of ports is specified (the examples above show the default values). If you deploy the flow to multiple execution groups, you will notice that the port number is incremented for successive deployments. The message flow deployed to the first execution group would receive requests on port 7800 (by default), the next one on 7801 etc, up to the specified limit of 7842. In this scenario, you would typically use some intermediary router that listened on one port and then distributed the requests across the range of ports you are using. Figure 1 below shows a network dispatcher being used in a DMZ.

If you don't want the port to be dynamically allocated, you can specify a specific port using the command:

- `mqsichangeproperties YOURBROKER -e YOUREXECUTIONGROUP`, followed by:
- `-o HTTPConnector -n explicitlySetPortNumber N`, where N is an integer in the range 0 to 65536

In this case the connector will not start if the specified port is already busy. The port used for the execution group is "sticky" -- if the broker is shut down, then it will use the same port number after restart.

Figure 1. Network Dispatcher HTTP input scenario



The network dispatcher owns the single external IP address and port number, and fans out the requests to the execution groups. If one execution group goes down, then the requests are sent to the remaining execution groups. The network dispatcher can potentially front-end several brokers on different machines, thereby providing greater reliability and implicit failover capacity.

Selective processing

Roles and 'mustUnderstand' headers

A key SOAP concept is the *role* (SOAP 1.2 terminology, equivalent to an *actor* in SOAP 1.1). The idea is that an application (in our case, a message flow) is contracted to fulfil a named role whilst processing a SOAP message. In practice this means that the message flow handles any SOAP header blocks flagged for the attention of that role.

Terminology

- **W3C**: World Wide Web Consortium, the body which publishes the Web services standards.
- **SOAP**: W3C standard XML message format for Web services messages.
- **SOAP with Attachments (SwA)**: W3C standard for Web services that need to incorporate attachments such as image data in their messages.
- **SOAP domain**: Broker domain for working with Web services messages. The messages are represented in a message flow using the **SOAP domain logical tree**.
- **Web Services Description Language (WSDL)**: W3C standard for describing a Web service.
- **Multipurpose Internet Mail Extensions (MIME)**: Message format for multipart messages and is the underlying format for SOAP with Attachments and MTOM.
- **Message Transmission Optimization Mechanism (MTOM)**: The use of MIME to optimize the bitstream transmission of SOAP messages that contain significantly large base64Binary elements.

- **WS-Addressing**: W3C specification that describes how to specify identification and addressing information for messages. It provides a correlation mechanism and allows more then two services to interact.
- **mustUnderstand header**: SOAP header marked with a mustUnderstand attribute set to '1' (as specified for SOAP 1.1 and the WS-I Basic Profile) or 'true' (SOAP 1.2).
- **DMZ**: Subnetwork that manages the exposure of enterprise resources to the outside world. Only resources in the DMZ are visible to the outside world.

There are two predefined roles: *Initial Sender* and *Ultimate Receiver*:

- The SOAPRequest and SOAPAsyncRequest nodes always act as Initial Sender when sending a request.
- The SOAPRequest and SOAPAsyncResponse nodes always act as Ultimate Receiver when receiving a response (the Ultimate Receiver is responsible for processing the SOAP payload).
- The SOAPInput node can be configured with a specific role, but by default acts as Ultimate Receiver.

You can specify a SOAP role name on the **Advanced Properties** tab of the SOAPInput node. If you do this you are saying that your message flow is to act as a Web service provider with that role, meaning that your message flow will handle all SOAP headers flagged with that role.

The mechanism for enforcing the use of roles is the `mustUnderstand` attribute. A header with `mustUnderstand` set must be understood by an application fulfilling the specified role, or an exception is thrown. The SOAP nodes provide a mechanism for defining the headers that can be handled by your flow (and which must not therefore be rejected with an exception if they are specified as mustUnderstand). It is your responsibility to ensure that your message flow really does handle these headers and performs any expected processing on them.

The mustUnderstand headers are defined in two separate tables on the **Advanced** Properties tab. On the SOAPInput node, the specified headers relate to the input message. On the SOAPRequest and SOAPAsyncRequest nodes, they relate to the response message.

The `WSDL-defined SOAP headers` table is populated automatically from the WSDL. You can select any of these headers to be added to the mustUnderstand headers list. Likewise, you can add headers that were not defined in the WSDL by adding them in the `User-defined SOAP headers` table. The headers must be selected or "ticked" to take effect. If the message contains a mustUnderstand header that is present in the table but unticked, then an exception is still thrown.

RouteToLabel support

Another feature that can be used to simplify SOAP message processing is the RouteToLabel node.

By default, the property **Set destination list** on the Advanced tab is selected and tells the SOAPInput and SOAPAsyncResponse nodes to populate the following field in the Local Environment with the current SOAP Operation name:

```
LocalEnvironment.Destination.RouterList.DestinationData.labelname
```

This lets you connect the Out terminal of these nodes to a RouteToLabel node which will then propagate the message to an appropriately configured Label node. It is your responsibility as a

message flow designer to configure the `Label name` property of each Label node appropriately to match the SOAP operation name.

If your message flow includes more than one SOAPInput node, then operations with the same name received at different nodes will be routed to the same label. If this is not what you want, you must also define the `Label prefix` property so that all the labels are unique within the flow.

Routing (WS-Addressing)

This section describes the concepts of WS-Addressing as they apply to the SOAP nodes. Without WS-Addressing, a response to a Web service request is always returned to the sender. This is often exactly what you want, but is not very flexible. With WS-Addressing, address information is carried explicitly in the message itself, providing support for:

- More complex message exchange patterns, involving multiple endpoints. A response no longer has to go to the original sender, and separate destinations can be specified for normal responses and SOAP faults.
- Asynchronous and "extended duration" communication patterns.
- A message correlation mechanism, which is used by other protocols such as WS-ReliableMessaging.

In essence, WS-Addressing defines a number of SOAP headers that your message flow may need to inspect or define at runtime. Listing 1 shows an example SOAP Header using WS-Addressing headers. If WS-Addressing is enabled, then inbound WS-Addressing headers are processed specially and appropriate headers are generated for outbound messages.

Listing 1. SOAP message example: WS-Addressing Headers

```
<SOAP-ENV:Header>
    <wsa:To>http://localhost:6080//myAppServer/services/myService</wsa:To>
    <wsa:Action>myOperation1</wsa:Action>
    <wsa:MessageID>uuid:515704D6-0111-4000-E000-82267F000001</wsa:MessageID>
    <wsa:ReplyTo>
        <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:FaultTo>
        <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    </wsa:FaultTo>
</SOAP-ENV:Header>
```

You tell a SOAP node to use WS-Addressing by checking **Use WS-Addressing** on the **WS Extensions** properties tab. The property can be set on the SOAPInput node and the SOAPRequest node (it is not set by default). It is implicitly selected for the SOAPAsyncRequest node.

LocalEnvironment Overrides

When you need to set LocalEnvironment overrides, be careful to set the **Compute mode** to **LocalEnvironment and Message** on the Basic tab of the Compute node Properties.

You override a WS-Addressing header in ESQL using the LocalEnvironment. See Listing2 below.

WebSphere Message Broker supports two versions of WS-Addressing: the Submission version and the Final version (see Related topics). You should use one version or the other. This section assumes use of the Final version of the specification, so in the example above `wsa` would be defined as follows:

```
xmlns:wsa="http://www.w3.org/2005/08/addressing"
```

WS-Addressing defines the following key headers:

- wsa:To, wsa:ReplyTo, wsa:FaultTo, wsa:From
  the endpoints to receive the message and its replies
- wsa:MessageID, wsa:RelatesTo
  message correlators
- wsa:Action
  a URI indicating how the message should be processed

The `ReplyTo`, `FaultTo` and `From` headers are defined as EPRs. (The `To` header is a simple URI, but Message Broker allows an EPR to be used for this too in some circumstances.) An EPR is a standard way of specifying an endpoint and other associated information.

WS-Addressing also defines two special addresses:

- `"http://www.w3.org/2005/08/addressing/anonymous"`, which means send the response back over the connection that the request was received on. The response is returned to the sender.
- `"http://www.w3.org/2005/08/addressing/none"`, which means no message should be sent because all messages to this address will be discarded.

The following ESQL shows the WS-Addressing `To` field being set to a specific endpoint and the `ReplyTo` field being set to anonymous.

Listing 2. Setting WS-Addressing Headers

```
SET OutputLocalEnvironment.Destination.SOAP.Request.WSA.To =
      'http://localhost:7800/test1';
SET OutputLocalEnvironment.Destination.SOAP.Request.WSA.ReplyTo =
      'http://www.w3.org/2005/08/addressing/anonymous';
```

The WS-Addressing header information is set in the LocalEnvironment. If you attempt to set the information under SOAP.Header as follows, then the information will not be used:

```
DECLARE wsa NAMESPACE 'http://www.w3.org/2005/08/addressing';
SET OutputRoot.SOAP.Header.wsa:To = 'http://localhost:7800/test1'; -- DO NOT DO THIS
```

WS-Addressing makes use of the following folders under LocalEnvironment. Each folder contains fields applicable to the node that will use the information. If you need to see all the fields, select the checkbox **Place WS-Addressing headers into LocalEnvironment** on the node. If this checkbox is not ticked then the addressing information is processed and discarded when the message is received (at the SOAPInput node for a request, or at a SOAPRequest or SOAPAsyncResponse node for a response message).

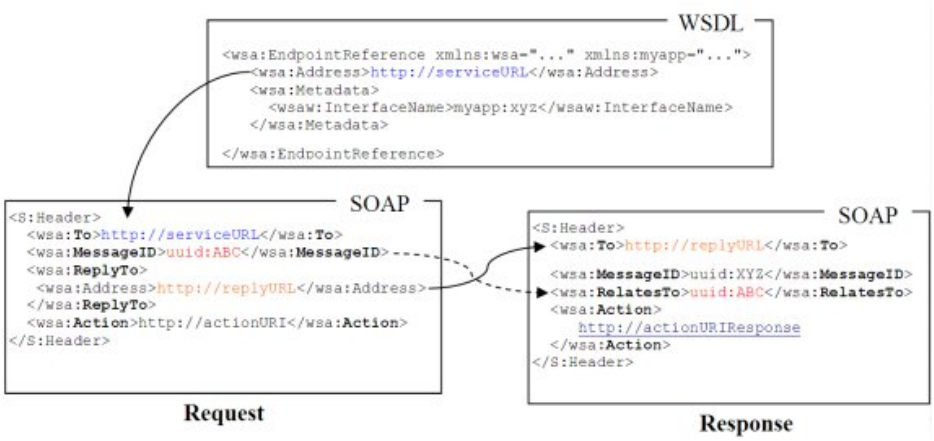
Table 1. WS-Addressing and LocalEnvironment

Folder Name	Populated by	Used by
SOAP.Input.WSA	SOAPInput (if checkbox ticked)	message flow
Destination.SOAP.Reply.WSA	message flow	SOAPReply
Destination.SOAP.Request.WSA	message flow	SOAPRequest (outbound), SOAPAsyncRequest
SOAP.Response.WSA	SOAPRequest (inbound), SOAPAsyncResponse (if checkbox ticked)	message flow

If the `ReplyTo` field is an explicit (non-anonymous) EPR then the response is sent to that endpoint using a new connection. If the `ReplyTo` field has the special value `anonymous` (the default) then the response is sent to the originator on the same connection (see Figure 5 later in this article).

Figure 2 shows how an EPR (EndpointReference) in WSDL relates to actual WS-Addressing headers used by SOAP at runtime:

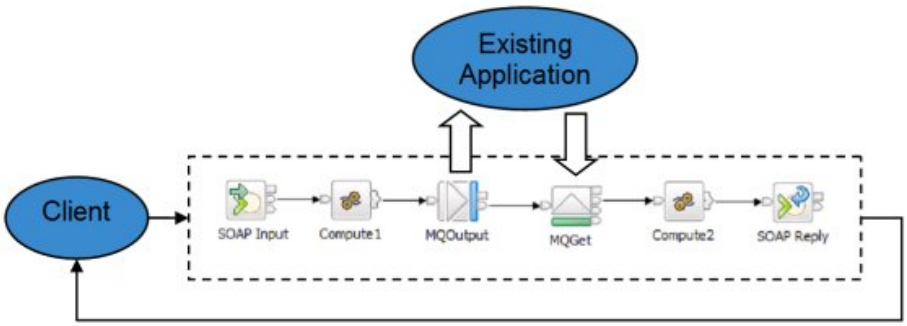
Figure 2. WS-Addressing Example with WSDL binding



The next section describes the use of WS-Addressing in the basic provider and consumer scenarios described in Part 1 of this series.

WS-Addressing use - provider scenario

Figure 3. Provider Scenario



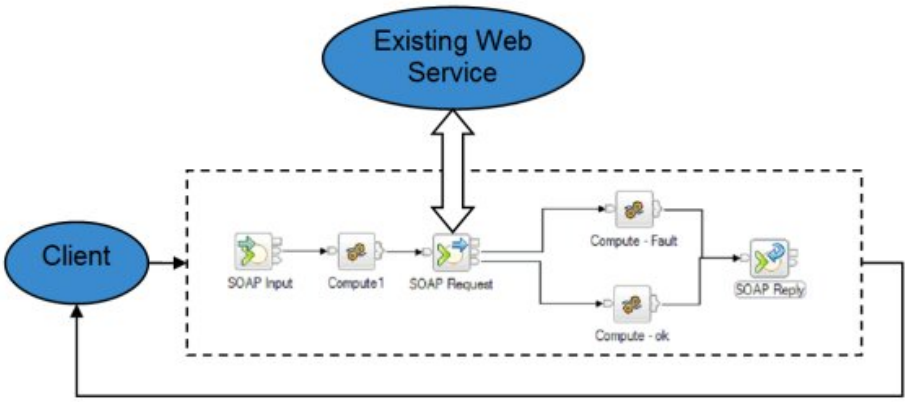
If WS-Addressing is not enabled then any WS-Addressing headers in the SOAP message received at SOAPInput are treated just like regular headers and propagated as children of SOAP.Header in the logical tree. (There is one special case. If the headers were explicitly defined as mustUnderstand headers then an exception is returned unless you have explicitly added the WS-Addressing headers to the mustUnderstand header lists. This is because by deselecting `Use WS-Addressing` you have implicitly said that your flow does not understand these headers.)

If WS-Addressing is enabled then any WS-Addressing headers in the incoming SOAP message are removed from the message tree, indicating that they have been processed. (Note that these original input headers would not be *valid* for use in the output message.) However, if the node option `Place WS-Addressing Headers into LocalEnvironment` is selected then the headers are made available in the LocalEnvironment tree under the `SOAP.Input.WSA` folder.

Correct reply headers are always automatically generated for you. If you need to override one or more of these defaults then you can specify overrides in the LocalEnvironment. If WS-Addressing was enabled on the SOAPInput node then it necessarily applies to the associated SOAPReply node. The message flow can override any addressing information required in the `Destination.SOAP.Reply.WSA` folder of the LocalEnvironment tree. If this folder is empty or does not exist, then the node will generate appropriate default headers using the addressing information from the incoming message.

WS-Addressing use - consumer scenario

Figure 4. Consumer Scenario



If WS-Addressing is not enabled, then no addressing headers are added to the outgoing request message and no addressing headers are processed in the response message.

If WS-Addressing is enabled (as it always will be in the case of SOAPAsyncRequest) then the following headers are always generated: `Action`, `To`, `ReplyTo`, and `MessageID`. The data for these headers (and others if present) is taken from the LocalEnvironment under `Destination.SOAP.Request.WSA`. However, if this folder does not exist or does not contain all the required fields then default headers are generated. (WS-Addressing headers placed directly into the SOAP.Header folder of the SOAP domain tree will *not* be processed as WS-Addressing headers, and will be replaced on output.)

Once the response to the request is received, all WS-Addressing headers are removed. However, if the node option `Place WS-Addressing Headers into LocalEnvironment` is selected then the headers are made available in the LocalEnvironment tree under the `SOAP.Response.WSA` folder.

WS-Addressing Action and SOAPAction

When a SOAPInput node receives a Web service request, it needs to identify the corresponding WSDL operation. There are four mechanisms for specifying the operation:

1. **SOAPAction** -- an HTTP transport header. The SOAPAction is optional, and often set to the empty string "". If specified it can be set to the SoapAction value in the WSDL (if any), or the operation name itself, and is saved in the logical tree under `HTTPInputHeader`.
2. **wsa:Action** -- a WS-Addressing header (we use the wsa: prefix as a shorthand for WS-Addressing). wsa:Action is mandatory if WS-Addressing headers are used in the message. If a wsa:Action is defined on the operation in the WSDL portType then you must use this value. If the wsa:Action attribute is not defined in the WSDL then you must use a default value derived from the WSDL as defined in Web Services Addressing 1.0 - WSDL Binding. If the node option `Place WS-Addressing Headers into LocalEnvironment` is selected, then wsa:Action is saved in the LocalEnvironment.
3. **SOAP payload** -- The name and namespace of the first child element below the SOAP Body. The SOAP payload is saved in the logical tree under SOAP.Body.
4. **Content-Type** -- Can specify an action parameter (in SOAP 1.2). This will override SOAPAction if both are present.

In practice:

- The SOAPInput node can usually determine the operation by looking at the SOAP payload, without needing SOAPAction or wsa:Action.
- If you're using WS-Addressing, you would typically set SOAPAction to the empty string and set wsa:Action as explained above.
- If you're not using WS-Addressing, and you want or need to set SOAPAction explicitly then use the WSDL soapAction if it is defined, otherwise the WSDL operation name.

There is one case where the operation cannot be deduced from the payload. This occurs if a document-style WSDL defines two or more operations with the same payload. In this case, the sender must specify a SOAPAction or wsa:Action. If the sender is using WS-Addressing then the preferred approach would be to set wsa:Action as described earlier and to set SOAPAction to "".

> The operation is always checked against the SOAP payload. If SOAPAction and/or wsa:Action are used, they must be consistent with the SOAP payload and with each other.

If a SOAPAction or wsa:Action is specified in the WSDL, this is the value that should be used.

If you need to *set* the wsa:Action or SOAPAction for a Web service request:

- If WS-Addressing is being used, a default `wsa:Action` is constructed based on information in your WSDL as described in the WS-Addressing specifications (see Related topics).

Alternatively you can set `wsa:Action` explicitly via the LocalEnvironment. For example, the following ESQL can be used prior to a SOAPRequest or SOAPAsyncRequest node to set the `wsa:Action` to `'op1'`: `SET OutputLocalEnvironment.Destination.SOAP.Request.WSA.Action = 'op1';`

- By default, the HTTP `SOAPAction` is set to the empty string "" if WS-Addressing is being used or if no specific value is defined in the WSDL. Alternatively you can set `SOAPAction` explicitly via the appropriate HTTP header. For example, the following can be used prior to a SOAPRequest or SOAPAsyncRequest node to set the `SOAPAction` to `"op1"`: `SET OutputRoot.HTTPRequestHeader.SOAPAction = '"op1"';`. The value has been explicitly quoted. Most endpoints accept a SOAPAction value without quotes, but the SOAP 1.1 specification does require the value to be enclosed in double quotes and some endpoints insist on this.

Error handling

This section describes those aspects of error handling that are unique to the SOAP nodes.

The SOAP specifications define the concept of a SOAP Fault as a special type of Web service message indicating that an error has occurred. In normal use, the SOAPRequest and SOAPAsyncResponse nodes should *expect* to receive SOAP Faults. This will occur whenever the Web service is unable to fulfil the request sent to it, by the SOAPRequest or SOAPAsyncRequest node respectively. (Incidentally, you would *not* expect a SOAPInput node to receive SOAP fault messages, or the SOAPAsyncResponse node to receive SOAP request messages.)

In order to make error handling more convenient, the SOAPRequest and SOAPAsyncResponse nodes therefore have an additional `Fault` terminal to which SOAP Faults will be routed. You should connect this terminal if you want your message flow to take specific action when a request fails.

Other exceptions are still directed to the regular `Failure` terminal, which is also present on these nodes.

The SOAPInput and SOAPAsyncResponse nodes are real "input" nodes, in the sense that they can start a message flow. They therefore also have a `Catch` terminal.

The use of the `Failure` and `Catch` terminals follows the usual message flow error handling principles. In brief:

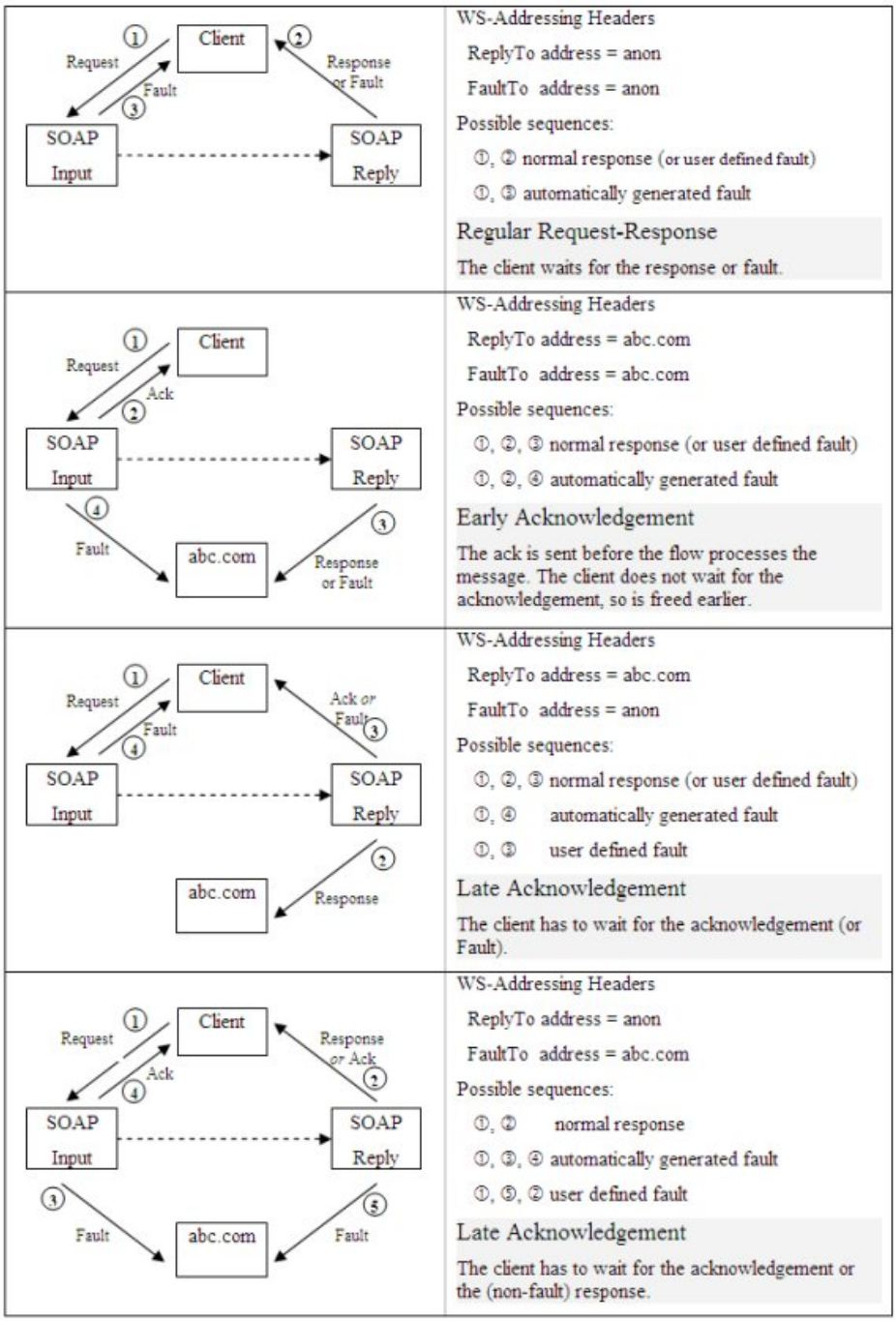
- The node propagates a message down the `Failure` terminal if this is connected and a failure is detected during processing within the node.
- The node propagates a message down the `Catch` terminal if this is connected and an exception is thrown downstream of the Out terminal or the Fault terminal.
- If the SOAPInput node `Catch` terminal is not connected then a Fault is sent back to the sender. (In the case of SOAPAsyncResponse, if the catch terminal is not connected the exception is logged and the message discarded as there is no mechanism available to automatically send a Fault back to the originating client.)

Also note that:

- If the `Catch` terminal is not connected (or an exception occurs downstream of a connected `Catch` terminal) then the SOAPInput node can return a SOAP Fault directly to the client, indicating a SOAP-specific error such as an invalid WSDL operation. The SOAP Fault is automatically populated with an exception list describing the problem. However, if the property `Send failures during inbound SOAP processing to failure terminal` is selected (on the `Error Handling` tab) then the fault is sent to the Failure terminal instead, allowing you to do additional processing such as logging. (Note that this only applies to errors caught within the input node. Once the message has been propagated to the Out terminal, it will never be propagated to the Failure terminal.)
- Timeouts can be defined on the `HTTP Transport` tab of the SOAPInput, SOAPRequest and SOAPAsyncRequest nodes. For example, if you know that the failure to return a SOAP reply or to receive a SOAP response within a specific interval means that an application has failed, then you can adjust the timeouts accordingly to prevent your message flow hanging unnecessarily.
- WS-Addressing allows specific end points to be specified as the destination for both normal responses and faults. These end points can be different from the original sender, and different from each other.

Figure 5 elaborates the last point and shows the message sequences that are possible for each combination of ReplyTo and FaultTo for a request using WS-Addressing. In each case, a client sends a request message and specifies whether the responses (both normal and fault) should be returned to the client (`'anon'`) or a separate end point (here, `'abc.com'`).

Figure 5. Fault handling with WS-Addressing



Debugging

You may need a test client to exercise your SOAP message flows. Options include:

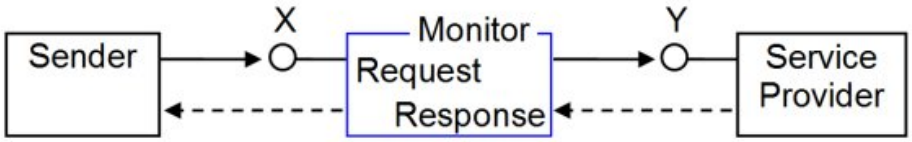
- WebSphere Message Broker Test Client
- A separate message flow using a SOAPRequest or HTTPRequest node
- A standalone client program such as Apache HTTPClient (see Related topics)

You may also need a network monitor to view the actual bitstream for messages sent and received. This could be particularly useful if you are working with MTOM or SwA messages. Suitable monitors include (see Related topics):

- Axis tcpmon
- Fiddler
- NetTool

In fact you can use Fiddler or NetTool as standalone clients too. These monitors work by intercepting network traffic. You tell the monitor to listen on one address (lets call it `X`) and to route all messages to another (lets call it `Y`). When you subsequently send a message to `X` it is routed to `Y` and then the reply (from `Y`) is routed back to you. The monitor displays the network traffic in both directions.

Figure 6. Interposing a Network Monitor



This works well *unless* you are using WS-Addressing. In the example above the real destination for the message is `Y`. However, if you set the WS-Addressing `To` field to `Y` then the message will bypass your monitor; whilst if you set the `To` field to `X` then the message may be rejected at `Y` (because the endpoint is incorrect).

A solution is to explicitly override the logical `To` address and the transport `To` address separately in the LocalEnvironment. In the example above you would set the transport address to `X` and the endpoint address to `Y`. For instance, before a SOAPRequest or SOAPAsyncRequest node:

```
SET OutputLocalEnvironment.Destination.SOAP.Request.Transport.HTTP.WebServiceURL =
    'http://X'; -- URI for X
SET OutputLocalEnvironment.Destination.SOAP.Request.WSA.To =
    'http://Y'; -- URI for Y
```

or, before a SOAPReply node making an async response:

```
SET OutputLocalEnvironment.Destination.SOAP.Reply.Transport.HTTP.AsyncReply.WebServiceURL=
    'http://X'; -- URI for X
SET OutputLocalEnvironment.Destination.SOAP.Reply.WSA.To =
    'http://Y'; -- URI for Y
```

As always, when you want to set LocalEnvironment overrides, set the **Compute mode** to **LocalEnvironment and Message** or **All** on the **Basic** tab of the Compute node Properties.

Conclusion

This series of articles described the new SOAP nodes, introduced in WebSphere Message Broker V6.1 as part of the long-term WebSphere commitment to Web services and SOA. The nodes

provide a framework for implementing Web services scenarios, including support for the next generation Web services standards WS-Addressing and WS-Security. You should now have an understanding of how the SOAP nodes behave and how you can configure and use them in common scenarios.

Related topics

- SOAP nodes in WebSphere Message Broker V6.1, Part 1: SOAP node basics
  This four-part series describes the SOAP nodes, the logical tree for the new SOAP domain, and explains their configuration and runtime behavior. In Part 1, you learn about the basic use of the nodes.
- SOAP nodes in WebSphere Message Broker V6.1, Part 2: SOAP domain logical tree
  Part 2 describes the new logical tree format used by the SOAP domain.
- SOAP nodes in WebSphere Message Broker V6.1, Part 3: Configuration details
  Part 3 shows you how to configure SOAP nodes using Web Services Description Language (WSDL).
- What's New in WebSphere Message Broker V6.1
  This article introduces the major enhancements WebSphere Message Broker V6.1, provides references to related resources, and describes technical aspects of V6.1 of interest to architects, message flow designers, and developers.
- W3C specifications for SOAP, SOAP with Attachments and MTOM, WSDL, and WSDL 1.1 Binding Extension for SOAP 1.2
- W3C specifications for WS-Addressing, including the Submission version and the Final version, which includes Core, SOAP Binding, and Metadata. WebSphere Message Broker supports both versions, but defaults to the Final version.
- WS-I specifications for the Basic Profile and Attachments Profile.
- WebSphere Message Broker product page
  Product descriptions, product news, training information, support information, and more.
- WebSphere Message Broker information center
  A single Web portal to all WebSphere Message Broker V6 documentation, with conceptual, task, and reference information on installing, configuring, and using your WebSphere Message Broker environment.
- WebSphere Message Broker documentation library
  WebSphere Message Broker specifications and manuals.
- Redbook: Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB
  Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying e-business applications. This Redbook shows you how to use WebSphere ESB together with WebSphere Message Broker to implement an ESB within an SOA. Includes scenario to demonstrate design, development, and deployment.
- Technical books from IBM Press
  Convenient online ordering through Barnes & Noble.

© Copyright IBM Corporation 2009
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)