BEA SALT™

Administration Guide

Version 1.1
Document Revised: August 16, 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Introduction

Configuring BEA SALT

Data Mapping and Conversions

Monitoring and Tuning Web Services

Introduction to Using Plug-ins with BEA SALT

Interoperability Considerations

Troubleshooting

CHAPTER 1

Introduction

This section contains the following topics:

Overview

BEA SALT(Service Architecture Leveraging Tuxedo) is a separately licensed product that runs on top of Tuxedo. BEA SALT exposes existing Tuxedo services as standard Web services and provides access points to Tuxedo services through SOAP over HTTP/S protocol.

In addition to basic Web service protocols, BEA SALT complies with most primary Web services specifications: WS-ReliableMessaging and WS-Addressing, SOAP 1.1, SOAP 1.2, and WSDL 1.1, allowing BEA SALT to interoperate with other Web service products and development toolkits. With BEA SALT, you can easily export existing Tuxedo services as Web services without having to perform any programming tasks.

Preparing to Install BEA SALT

Before installing BEA SALT, ensure the following prerequisites are met.

- You have successfully installed Tuxedo 8.1 or Tuxedo 9.1 server components. For more server component information, see Tuxedo 8.1 install sets and Tuxedo 9.1 install sets.

Notes: For Tuxedo 8.1 (Windows), rolling patch 268 or above is required.

For Tuxedo 8.1 (UNIX), rolling patch 265 or above is required.

For Tuxedo 9.1 (Windows and UNIX), rolling patch 003 or above is required.

The rolling patch can be found on the product CD. You can also contact BEA Support for the latest rolling patch version.

Components for Administering BEA SALT

The following components are required to configure SALT and deploy and invoke Tuxedo services using SALT.

- SALT Configuration File
- GWWS Server
- WSDL

SALT Configuration File

BEA SALT uses configuration files in an XML format to define necessary deployment information for exporting Tuxedo services as Web services, which includes a Tuxedo service list, WS-ReliableMessaging Policy information, and SOAP protocol binding information. Each configuration file can define multiple GWWS server instances to enable failover capability and generate multiple port information in the WSDL document. The SALTconfiguration file leverages the Tuxedo Service Metadata Repository for Tuxedo service contract information.

The SALT configuration file is a single root XML file. the root element is `<Configuration>`, which has four sub-elements, `<Servicelist>`, `<Policy>`, `<System>` and `<WSGateway>`. For more information, see “Configuring BEA SALT” on page 2-1.

GWWS Server

BEA SALT provides a Tuxedo system gateway, the GWWS server, which handles Web service SOAP messages over HTTP/S protocol. GWWS acts as a Tuxedo gateway process and is managed in the same manner as general Tuxedo system servers. The GWWS server is a configuration-driven program. Each SALT configuration file defines one or more GWWS instances. Each SALT configuration file maps to a single SOAP/WSDL Web service object in

the component model of the WSDL specification. Each GWWS process defined in the configuration file serves as a low-level port object of the service object. Each Tuxedo application service defined in the configuration file is treated as a WSDL service object operation. For more information, see "Configuring BEA SALT" on page 2-1.

The GWWS server handles Web service requests in the following manner.

1. The GWWS server parses SOAP request messages and converts them into Tuxedo typed buffers.
2. The GWWS server dispatches the Tuxedo typed buffers to corresponding Tuxedo services.
3. The Tuxedo service returns Tuxedo response typed buffers.
4. The GWWS server converts the Tuxedo response typed buffers to SOAP response messages and sends it back to the client.

Failover with GWWS Server

One configuration file is used to represent a particular GWWS server or a group of failover GWWS server. Configuring multiple GWWS instances within the SALT configuration file enables you to deploy failover capability over multiple GWWS instances.

Handling Custom Buffers

SALT provides a plug-in mechanism for converting SOAP XML messages and Tuxedo custom typed buffers. You can validate the SOAP message against your own XML Schema definition, allocate custom typed buffers, and parse data into the buffer and other operations. For more information see, "Introduction to Using Plug-ins with BEA SALT" on page 5-1.

WSDL

BEA SALT generates a WSDL document according to the SALT configuration file. The generated WSDL document describes the capability of a particular GWWS process or a group of failover GWWS processes. There are two ways to obtain the WSDL document. You can:

- use the `tmwsdlgen` WSDL document file generating utility.
- use the GWWS server HTTP download service

The WSDL generating process needs to import the following resources for WSDL document content:

- Tuxedo Service contract information defined in the Tuxedo Service Metadata Repository for abstract service descriptions in the WSDL document.

  Note: The Tuxedo Service Metadata Repository is a prerequisite for using SALT. You must define your Tuxedo application services in the Tuxedo Service Metadata Repository before the service can be exported as a Web service.

- WS-ReliableMessaging Policy file for WSDL document attachments.

  Defines the WS-Reliable Messaging for the SALT configuration file.

- SOAP binding style and end point address information.

The generated WSDL document can interoperate and integrate with the your Web services development tools or it can be published to a UDDI server.

Scenario for Deploying and Invoking a Tuxedo Service Using SALT

The following is a typical scenario for deploying and invoking a Tuxedo service using SALT:

1. Compose one or more SALT configuration files.
2. Define Tuxedo application services using the Tuxedo Service Metadata Repository.
3. Compose the Tuxedo UBBCONFIG file to include:
   - One or more SALT GWWS server gateway instances
   - TMMETADATA instances
   - Other necessary system and application servers.
4. Boot your Tuxedo application.
5. Client downloads the WSDL document from the specific GWWS URL.
6. Client generates stub-code from the WSDL document using a SOAP development toolkit.
7. Compose the client program.
8. Run the client to invoke the Web service with SOAP messages.
9. The GWWS server translates the SOAP message into Tuxedo typed buffer, dispatches the Tuxedo service and gets the reply. The GWWS then translates the reply into a SOAP message and sends it back to the client.

See Also

- UBBCONFIG(5)
- Tuxedo Service Metadata Repository
- GWWS, `tmwsdlgen`, `wsadmin`

CHAPTER 2

Configuring BEA SALT

This section contains the following topics:

Creating a BEA SALT Configuration File

The SALT configuration file uses XML format to define necessary deployment information for exporting Tuxedo services as Web services. This deployment information includes the following definitions: a Tuxedo service list, WS-ReliableMessaging Policy information, and SOAP protocol binding information. The deployment information generates a corresponding WSDL document. Multiple GWWS instances enable failover capability and generate multiple port information in the WSDL document.

SALT configuration leverages Tuxedo Service Metadata Repository for Tuxedo service contract information. BEA SALT accesses the Tuxedo Service Metadata Repository system server, TMMETADATA, provided by the local Tuxedo domain and gathers corresponding service contract information to generate the SALT WSDL document.

Sample SALT Configuration File

In Listing 2-1, the namespace of this version configuration XML file is defined with a fixed URL string "`http://www.bea.com/Tuxedo/Salt/200606`", which is used to identify the SALT version associated with the configuration file.

Listing 2-1 Sample SALT Configuration File

```
<? xml version="1.0" encoding="UTF-8"?>
<Configuration xmlns="http://www.bea.com/Tuxedo/Salt/200606" >
       <Servicelist  id="simple">
              <Service name="toupper" />
              <Service name="tolower" />
       </Servicelist>
       <Policy>
              <RMPolicy>RMpolicy.xml</RMPolicy>
       </Policy>
       <System>
              <Certificate>
                     <PrivateKey>cert.pem</PrivateKey>
              </Certificate>
</System>
<WSGateway>
              <GWInstance  id="GW1">
                     <HTTP address="//my server" />
                     <HTTPS address="//my server" />
              </GWInstance>
       </WSGateway>
</Configuration>
```

SALT Configuration Format

Each BEA SALT configuration file is composed of the following four XML format elements:

- **<Servicelist>**

  Defines a list of Tuxedo services to be exposed as Web services.

- **<Policy>**

  Defines a global WS-ReliableMessaging Policy that is applied to all listed Tuxedo services.

- **<System>**

  Defines system-level parameters

- **<WSGateway>**

  Defines one or more SALT GWWS processes that export the specified Tuxedo services

The syntax for each of these for elements is listed in Table 2-1.

Configuring Reliable Messaging Policy

BEA SALT supports the WS-ReliableMessaging Policy. The <RMPolicy> sub-element is used to define the policy.

Defining System Parameters

- <System> defines all system-level or global parameters related to Tuxedo and the operating system. The <System> element supports the following sub-elements:
- <Certificate> element includes private key and certificates parameters necessary for setting up HTTPS connections.
- <<Plugin> defines information for the GWWS server plug-in framework to support Tuxedo custom typed buffers.
- <LogLevel> specifies the log level for information printing during the SALT configuration parsing process.

Defining the GWWS Gateway

The GWWS gateway must be defined in the SALT configuration file and defined as a Tuxedo server in the UBBCONFIG file. For more information, see “Configuring Security” on page 2-23.

After defining the GWWS gateway through the SALT configuration file, boot the GWWS server using tmadmin or tmboot. When the GWWS gateway is initiated, it loads the specified SALT

configuration file, validates the XML configuration file, loads corresponding Tuxedo service contract information from Tuxedo Service Metadata Repository, and loads WS-ReliableMessaging policy definition file.

At runtime, the GWWS server reloads the configuration dynamically. You can also download the WSDL document from the GWWS server which is based on the latest configuration file.

SALT Configuration File Element Syntax

Table 2-1 describes the syntax used in each of these four elements.

Table 2-1 SALT Configuration Elements

Element	Description

<Servicelist>	Lists Tuxedo services to be exported as Web services.
	
	Attribute: id
	
	The name of the service list. This name is used in the generated WSDL document to compose the name of the wsdl:PortType object. The value of this attribute must be a [1,32] length string.
	
	Sub-element: <service>
	
	Specifies one Tuxedo service to be exported.
	
	Zero <Service> elements for <Servicelist> is a valid configuration which means no Tuxedo service is exported.
	
	<service> has the following attribute:
	
	Attribute: name
	
	The unique name of a service. This name is the key string inquiry for the service information defined in the Tuxedo Service Metadata Repository. The value of this attribute must be a [1,15] length string. The name attribute must match the Tuxedo Service Metadata Repository service keyword.
	
	Note: Tuxedo Service Metadata Repository provides two system-level keywords, service and tuxservice for Tuxedo service references. This provides the ability to define multiple service contract entries for actual the same Tuxedo service.
	
	The following is an example:
	
	`<Servicelist id="simple">`
	`<Service name="toupper" />`
	`<Service name="tolower" />`
	`</Servicelist>`

Table 2-1 SALT Configuration Elements

Element	Description

<Policy>	Specifies WS Policies applied to the SALT Gateway.
	
	Note: BEA SALT 1.1 can only define the WS-ReliableMessaging Policy.
	
	Attribute: NA.
	
	Sub-element: `<RMPolicy>`
	
	Specifies a WS-ReliableMessaging Policy. This element is optional.
	
	Text enveloped with this element indicates a valid file path for a WS-ReliableMessaging policy file on the local file system. The policy definition file must comply with WS-Policy and WS-ReliableMessaging specifications.
	
	The following is an example:
	`<Policy>`
	`<RMPolicy>RMpolicy.xml</RMPolicy>`
	`</Policy>`

Table 2-1 SALT Configuration Elements

Element	Description
<System>	Defines system-level or global parameters related to Tuxedo and the operating system. **Attribute:** NA. **Sub-element:** <Certificate> Specifies certificate information for HTTP over SSL connections. This element has no attribute. Sub elements are needed for necessary SSL certificate settings. <Certificate> has the following sub-elements: .../<PrivateKey> Specifies the PEM format private key file. The key file path is specified as the text value for this element. If the <Certificate> element is present, this element is required. The server certificate is also in this private key file. .../<VerifyClient> Specifies if Web service clients are required to send a certificate through HTTP over SSL connection. This element is optional. The text "true" and "false" are the only valid values for this element. "false" is the default value if not specified. .../<TrustedCert> Specifies the file name of the trusted PEM format certificate files. This element is optional. .../<CertPath> Specifies the local directory where the trusted certificates are located. This element is optional. If <VerifyClient> is true, or if WS-Addressing is used with SSL, at least one valid file must be specified for this element. If <VerifyClient> is true, or if WS-Addressing is used with SSL, you must set either <TrustedCert> or <CertPath>. The following is an example: <System> <Certificate> <PrivateKey>cert.pem</PrivateKey> </Certificate> </System>

Table 2-1 SALT Configuration Elements

Element	Description

	Sub-element: `<Plugin>`
	
	Specifies SALT plug-in interfaces. In the BEA SALT 1.1 release, plug-in interfaces are only used to extend data conversion between SOAP messages and Tuxedo custom typed buffers. For more information, see Using Plug-ins with BEA SALT in *Programming Web Services*.
	
	`<Plugin>` has the following sub-element:
	
	`.../<Interface>`
	Each `<Interface>` sub-element specifies a separate plug-in implementation. There can be zero or more `<Interface>` sub-elements for `<Plugin>` to indicate different plug-in implementations.
	
	<Interface> has the following sub-elements:
	
	`.../<ID>`
	Specifies the identifier of a plug-in interface. The identifier of a plug-in interface indicates the purpose of the plug-in. It can also be understood as the type of a plug-in. Multiple plug-in interfaces can be defined with the same <ID> value. The value of <ID> must be predefined by SALT. This element is required for a plug-in interface.
	
	In the BEA SALT 1.1 release, the only valid identifier value is `P_CUSTOM_TYPE`, which is used to identify the plug-in interfaces for Tuxedo custom typed buffer data conversion.
	
	`.../<Name>`
	Specifies the unique name of a plug-in interface. The name must be unique among the plug-in interfaces with the same <ID> value. The name may have a different meaning within *different* plug-in <ID> values. This element is required for a plug-in interface.
	
	In BEA SALT 1.1, for `P_CUSTOM_TYPE` typed plug-in interfaces, the name value must be the Tuxedo custom buffer type name.
	
	`.../<Library>`
	Specifies the shared library file that has the functions implemented for this plug-in interface. This element is required for a plug-in interface.
	
	`.../<Params>`
	Specifies a string value that can be passed to the initialization function of the plug-in interface. This element is optional for a plug-in interface.

Table 2-1 SALT Configuration Elements

Element	Description
	The following is an example: <System> <Plugin> <Interface> <ID>P_CUSTOM_TYPE</ID> <Name>MYBUFT</Name> <Library>Mybuft.so</Library> </Interface> </Plugin> </System>
	Sub-element: `<LogLevel>` Specifies the log level for information printing during configuration parsing. The value range is [1-8]: • 1-Minimal parsing information printed • 2-General error messages printed • 4-Warning message printed • 8-All parsing information printed Other numbers are reserved for future release. If this element is not specified, the default setting is 2.

Table 2-1 SALT Configuration Elements

Element	Description
<WSGateway>	Defines run time parameters for GWWS gateway instances. These instances share the same <Servicelist>, <Policy> and <System> configuration information. **Attribute:** NA. **Sub-element:** <GWInstance> Configure runtime parameters for one GWWS gateway. This element can be specified [1, 1024] times for <WSGateway>, in other words, at least one GWWS process and at most 1024 GWWS processes can be specified in one SALT configuration file. <GWInstance> has the following attribute: **Attribute**: id The GWWS instance identifier. The ID value must be unique within the Tuxedo domain. The value of this attribute must be a [1,12] length string. <GWInstance> has the following sub-elements: ...<HTTP> or <HTTPS> Specifies the HTTP or HTTP over the GWWS listening SSL endpoint address. At least one element, either <HTTP> or <HTTPS> must be specified for <GWInstance>. <HTTP>or <HTTPS> has the following attribute: **Attribute**: address The network address. The string value can be in one of the following two formats: //<ipaddress>:<portnum> or //<hostname>:<portnum>

Table 2-1 SALT Configuration Elements

Element	Description
	...<property> Specifies runtime properties for the GWWS process. This element is optional. Different properties can be set by specifying multiple <property> elements. <property> has the following attributes: **Attribute**: name The property name. **Attribute**: value The property value. Valid GWWS properties are: **Property**: max_content_length This property enables the GWWS server to deny the HTTP requests when the content length is larger than this property setting. If not specified, the GWWS server does not check for it. The string value can be one of the following three formats: 1. Integer number in bytes, no suffix means the unit is bytes 2. Float number in kilobytes, the suffix must be K 3. Float number in megabytes, the suffix must be M The equivalent byte size value must be in [1 byte, 1G byte] range. **Property**: thread_pool_size Defines the GWWS process thread pool size. The value must be in [1, 1024]. If not specified, default value 16 is used. **Property**: timeout Defines the network time-out value, unit: second. The value must be in [1, 65535]. If not specified, default value 300 is used.

Table 2-1 SALT Configuration Elements

Element	Description
	Property: `max_backlog` Specifies the backlog listen socket value. It controls the maximum length of the queue of pending connections by operating system. The value range for this property is [1-255]. The default value is 16. Generally no tuning is needed for this value. The following is an example: <WSGateway> <GWInstance id="GW1"> <HTTP address="//myhost:8001" /> <HTTPS address="//myhost:8002" /> <property name=" thread_pool_size" value="40" /> </GWInstance> </WSGateway>

Accessing Service Definitions from the Tuxedo Service Metadata Repository

BEA SALT leverages the Tuxedo Service Metadata Repository to define service contract information for Web Services. Service contract information of all listed Tuxedo services is obtained by accessing the Tuxedo Service Metadata Repository system service provided by the local Tuxedo domain. SALT calls TMMETADATA system server under the following scenarios:

- During GWWS server startup, SALT calls the Tuxedo Service Metadata Repository to retrieve all Tuxedo service definitions according to the services listed in the SALT configuration file.
- Initiation of the `configreload` command used in `wsadmin`. In this case, the GWWS server reloads the information from TMMETADATA.
- Invoking `tmwsdlgen` to generate a WSDL file causes SALT to call TMMETADATA.

The following topics provide SALT-specific interpretations of the Tuxedo Service Metadata Repository keywords and parameters:

- Defining Service-level Keywords for BEA SALT
- Defining Service Parameters for BEA SALT

Defining Service-level Keywords for BEA SALT

The following Tuxedo Service Metadata Repository service-level keywords have specific SALT interpretations for handling Web Service processing.

Note: Service-level keywords not specified in Table 2-2 have no special semantics for BEA SALT and are ignored when the Tuxedo Service Metadata Repository is loaded by SALT.

Table 2-2 BEA SALT Handling of Service-level Keywords in Tuxedo Service Metadata Repository

Service-level Keyword	Keyword Abbreviation	BEA SALT Interpretation
`service`	`sv`	The unique key value to distinguish one service from another in the Tuxedo Service Metadata Repository. This is the value to reference in the SALT configuration file.
`tuxservice`	`tsv`	The actual Tuxedo service name. BEA SALT invokes the Tuxedo service defined with this keyword. If no value is specified in `tuxservice`, then the value will be the same as the value in the `service` keyword.
`servicetype`	`st`	BEA SALT uses this keyword value to determine the service message exchange pattern for the specified Tuxedo service. BEA SALT maps the service with the Web Service message exchange pattern (MEP). The following values specify mapping rules between the Tuxedo service types and Web Service MEP: • `service` corresponds to request-response MEP • `oneway` corresponds to oneway request MEP • `queue` corresponds to request-response MEP
`export`	`ex`	BEA SALT ignores this value. If the service is specified in the SALT configuration `<Servicelist>`, it will be exposed regardless of the value specified in `export`.

Table 2-2 BEA SALT Handling of Service-level Keywords in Tuxedo Service Metadata Repository

Service-level Keyword	Keyword Abbreviation	BEA SALT Interpretation
`inbuf`	`bt`	Specifies the input buffer type. SALT provides default data representation and conversion between SOAP XML payload and the following Tuxedo buffer types: • STRING (case sensitive) • CARRAY • XML • MBSTRING • VIEW • VIEW32 • FML • FML32 • X_C_TYPE • X_COMMON • X_OCTET **Note:** If `inbuf` specifies any other type other than the previous buffer types, the buffer is treated as a custom buffer type.

Table 2-2 BEA SALT Handling of Service-level Keywords in Tuxedo Service Metadata Repository

Service-level Keyword	Keyword Abbreviation	BEA SALT Interpretation
`outbuf`	`BT`	Specifies the output buffer type. SALT provides default data representation and conversion between SOAP XML payload and the following Tuxedo buffer types: • STRING (case sensitive) • CARRAY • XML • MBSTRING • VIEW • VIEW32 • FML • FML32 • X_C_TYPE • X_COMMON • X_OCTET **Note:** If `outbuf` specifies any other type other than the previous buffer types, the buffer is treated as a custom buffer type.
`inview`	`vn`	Specify the view name used by the service if the service buffer type is VIEW, VIEW32, X_C_TYPE, or X_COMMON. BEA SALT requires that you specify the view name rather than accept the default `inview` setting.
`outview`	`VN`	Specify the view name used by the service if the service buffer type is VIEW, VIEW32, X_C_TYPE, or X_COMMON. BEA SALT requires that you specify the view name rather than accept the default `outview` setting.

Defining Service Parameters for BEA SALT

The Tuxedo Service Metadata Repository interprets parameters as sub elements encapsulated in a Tuxedo service typed buffer. Each parameter can have its data type, occurrences in the buffer, size restrictions, and other Tuxedo-specific restrictions.

- For VIEW, VIEW32, X_C_TYPE, or X_COMMON buffer types, each parameter of the buffer should represent a VIEW/VIEW32 structure member.
- For FML or FML32 buffer types, each parameter of the buffer should represent an FML/FML32 field element that may be present in the buffer.
- For STRING, CARRAY, XML, MBSTRING, and X_OCTET buffer types, the Tuxedo framework treats these buffers holistically. At most, one parameter is permitted for the buffer to define some restriction facets, such as buffer size threshold.
- For any custom type buffer, parameters facilitate describing details about the buffer type.
- For FML32 buffers that support embedded VIEW32 and FML32 buffers, embedded parameters provide that support.

Note: Parameter-level keywords not specified in Table 2-3 have no special semantics for BEA SALT and are ignored when the Tuxedo Service Metadata Repository is loaded by SALT.

Table 2-3 BEA SALT Handling of Parameter-level Keyword in Tuxedo Service Metadata Repository

Parameter-level Keyword	Abbreviation	BEA SALT Interpretation
`param`	`pn`	Specifies the parameter name. This keyword is required. • For VIEW, VIEW32, X_C_TYPE, or X_COMMON, specify the view structure member name in the `param` keyword. • For FML, FML32 typed buffers, specify the FML/FML32 field name in the `param` keyword. • For STRING, CARRAY, XML, MBSTRING, or X_OCTET, BEA SALT ignores the parameter definitions.
`type`	`pt`	Specifies the data type of the parameter. **Note:** BEA SALT does not support `dec_t` and `ptr` data types.

Table 2-3 BEA SALT Handling of Parameter-level Keyword in Tuxedo Service Metadata Repository

Parameter-level Keyword	Abbreviation	BEA SALT Interpretation
`subtype`	`pst`	Specifies the view structure name if the parameter type is VIEW32. For any other typed parameter, BEA SALT ignores this value. **Note:** The Tuxedo Service Metadata Repository allows this value to be empty if the parameter type if VIEW32. In this case the Tuxedo Service Metadata Repository uses a fixed value <viewname> as a default. This default value will not work for BEA SALT. You *must* specify the view name for BEA SALT to have a valid service definition.
`access`	`pa`	The general definition applies for this parameter.
`count`	`po`	The general definition applies for this parameter. For BEA SALT, the value for the `count` parameter must be greater than or equal to `requiredcount`.
`requiredcount`	`ro`	The general definition applies for this parameter. The default is 1. For BEA SALT, the value for the `count` parameter must be greater than or equal to `requiredcount`.
`size`	`pl`	This optional keyword restricts the maximum length of the parameter. It is only valid for parameter types: STRING, CARRAY, XML, and MBSTRING. If this keyword is not set, there is no maximum length restriction for this parameter. The value range is [0, 2147483647]

Creating a Policy File

To use WS-ReliableMessaging functionality, you must create a policy file and specify the policy file in the SALT configuration file. The format of the policy file is defined by the WS-ReliableMessaging Policy specification.

Listing 2-2 shows an RMAssertion syntax example.

Listing 2-2 RMAssertion Syntax

```
<wsrm:RMAssertion [wsp:Optional="true"]? ... >
      <wsrm:InactivityTimeout Milliseconds="xsd:unsignedLong" ... /> ?
      <wsrm:BaseRetransmissionInterval Milliseconds="xsd:unsignedLong".../>?
      <wsrm:ExponentialBackoff ... /> ?
      <wsrm:AcknowledgementInterval Milliseconds="xsd:unsignedLong" ... /> ?
      <beapolicy:Expires Expires="xsd:duration" ... /> ?
      <beapolicy:QOS QOS="xsd:string" ... /> ?
...
</wsrm:RMAssertion>
```

Sample Policy File

Listing 2-3 is shows a sample SALT policy file.

Listing 2-3 Sample Policy File

```
<?xml version="1.0"?>
<wsp:Policy wsp:Name="ReliableSomeServicePolicy"
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:beapolicy="http://www.bea.com/wsrm/policy">
<wsrm:RMAssertion>
      <wsrm:InactivityTimeout Milliseconds="600000" />
      <wsrm:AcknowledgementInterval Milliseconds="2000" />
      <wsrm:BaseRetransmissionInterval Milliseconds="500"/>
      <wsrm:ExponentialBackoff />
      <beapolicy:Expires Expires="P1D" />
      <beapolicy:QOS QOS="ExactlyOnce InOrder" />
</wsrm:RMAssertion>
</wsp:Policy>
```

All RM assertions are optional, and if not specified, the default value are used. The following definitions describe the RM assertion options.

RM Assertion Option	Description
`<wsrm:InactivityTimeout>`	Specifies the number of milliseconds, specified with the Milliseconds attribute, which defines an inactivity interval. After time has elapsed, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may terminate current sequence due to inactivity. The source endpoint can also use this parameter. Sequences never time out by default.
`<wsrm:AcknowledgementInterval>`	Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand-alone acknowledgement. There is no time limit by default.
`<wsrm:BaseRetransmissionInterval>`	Specifies the interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message. This value will apply to the GWWS server when it sends a response in an outbound sequence. The default value is 20000 milliseconds.
`<wsrm:ExponentialBackoff>`	Specifies that the retransmission interval is adjusted using the exponential backoff algorithm. This value applies to the GWWS server when it sends a response in an outbound sequence.

RM Assertion Option	Description
<beapolicy:Expires>	Specifies the amount of time after which the reliable Web service expires and does not accept any new sequence messages. This element has a single attribute, Expires, whose data type is an XML Schema duration type. For example, if you want to set the expiration time to one day, use the following: `< beapolicy:Expires Expires="P1D" />` The default value is `never expire`.
<beapolicy:QOS>	Specifies the delivery assurance. SALT supports the following assurances: • `AtMostOnce` - Messages are delivered at most once, without duplication. There is possibility that some messages may not be delivered. • `AtLeastOnce` – Every message is delivered at least once. There is possibility that some messages are delivered more than once. • `ExactlyOnce` – Each message is delivered exactly once, without duplication. • `InOrder` – Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances. The default value is “ExactlyOnce InOrder”.

Specifying a Policy File in the Configuration File

You must reference the WS-ReliableMessaging policy file at the end-point level in the SALT configuration file. The following entry in the SALT configuration file shows how to reference the WS-ReliableMessaging policy file.

```
<Policy>
     <RMPolicy>RMpolicy.xml</RMPolicy>
</Policy>
```

Note: Reliable Messaging in BEA SALT does not support process/system failure scenarios, which means SALT does not store the message in a persistent storage area. BEA SALT works in a *direct mode* with the SOAP client. Usually, system failure recovery requires business logic synchronization between the client and server.

Configuring the GWWS Gateway

The GWWS gateway provides the connection between the Tuxedo service and the Web service client. Two critical configurations must exist for the GWWS gateway process to reliably access the Tuxedo services.

- You must configure unique GWWS instances in the SALT configuration file.
- You must configure the GWWS instances as a Tuxedo server in the `*SERVERS` section of the UBBCONFIG file.

These two configurations provide the gateway connection that allows SALT to access Tuxedo services.

Configuring the Gateway as a Tuxedo System Server

Configure the GWWS gateway in the Tuxedo UBBCONFIG file to run as a Tuxedo system server. You can define multiple GWWS server instances concurrently in a Tuxedo UBBCONFIG file. Listing 2-4 lists part of the UBBCONFIG file showing how to define GWWS instances for a Tuxedo application.

Listing 2-4 GWWS Gateways Defined as Tuxedo Servers in the UBBCONFIG File *SERVERS Section

```
......
*SERVERS
GWWS SRVGRP=GROUP1 SRVID=10
CLOPT="-A -- -c saltconf_1.xml –i GW1"
GWWS SRVGRP=GROUP1 SRVID=11
CLOPT="-A -- -c saltconf_1.xml –i GW2"
GWWS SRVGRP=GROUP2 SRVID=20
CLOPT="-A -- -c saltconf_2.xml –i GW3"
......
```

In the UBBCONFIG file `*SERVERS` sections, you must specify the SALT configuration file and the ID of the GWWS instances defined in the SALT configuration.

For more information, see GWWS.

Note: Each Web service client is deemed a workstation client in Tuxedo; therefore, MAXWSCLIENTS must be configured with a valid number in UBBCONFIG for the node where the GWWS server is deployed.

Be sure that the TMMETADATA system server is set up in the UBBCONFIG file to start before the GWWS server boots. Because the GWWS server calls services provided by TMMETADATA, it must boot after TMMETADATA.

To ensure TMMETADATA is started prior to being called by the GWWS server, put TMMETADATA before GWWS in the UBBCONFIG file or use SEQUENCE parameters in *SERVERS definition in UBBCONFIG file.

Configuring Security

BEA SALT provides point-to-point security using SSL link-level security and uses the Tuxedo security framework for authentication. User profiles are passed via HTTP basic authentication specifications.

Setting Up SSL Link-Level Security

To set up link-level security using SSL, you must specify <Certificate> elements in the <System> parameter in the SALT configuration so that the GWWS gateway can handle HTTPS request. You can enable link-level security by specifying the HTTPS:// address in the SALT configuration file. For information about how to define <Certificate>, refer to "SALT Configuration Format" on page 2-3.

If the GWWS is configured to use SSL, SEC_PRINCIPAL_PASSVAR and SEC_PRINCIPAL_NAME *must* be configured for GWWS in the *SERVER definition in the UBBCONFIG file.

Setting Up HTTP Basic Authentication

BEA SALT uses the Tuxedo security framework for system authentication. The GWWS server supports Tuxedo user profile throughput from the Web service client via HTTP basic authentication protocol.

The GWWS gateway supports Tuxedo domain security configuration for the following two authentication patterns:

- Application password (APP_PW)
- User-level authentication (USER_AUTH)

The GWWS server passes the following string from the HTTP header of the client SOAP request for Tuxedo authentication.

```
Authorization: Basic <base64Binary of username:password>
```

The following is an example of a string from the HTTP header:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

In this example, the client sends the Tuxedo username `Aladdin` and the password `open sesame`, and uses this paired value for Tuxedo authentication.

- Using Application Password (APP_PW)

  If Tuxedo uses APP_PW, then the HTTP username value is ignored and the GWWS server only uses the password string as the Tuxedo application password to check the authentication.

- Using User-level Authentication (USER_AUTH)

  If Tuxedo uses USER_AUTH, then both the HTTP username and password value is used. In this case, the GWWS server does not check the Tuxedo application password.

  Note: `ACL` and `MANDATORY_ACL` are not supported for Web service clients, which means the Tuxedo system ignores any ACL-related configuration specifications. BEA SALT does not make group information available for Web service clients.

Validating the Configuration

BEA SALTcomponents validate the given SALT configuration according to the predefined XML Schema file. A valid SALT configuration file is a prerequisite for the GWWS server to start and for `tmwsdlgen` to generate the WSDL document file.

The SALT configuration schema file can be found at `$TUXDIR/udataobj/saltconfig_200606.xsd`.

BEA SALT uses the schema file to validate the SALT configuration file when:

- the GWWS server is booting. If the validation fails, the GWWS server shuts down and specific error information is sent to the ULOG file.
- the GWWS server is reloading the SALTconfiguration file at runtime. If the validation fails, the GWWS server stops reloading the new or updated configuration file and the original configuration remains effective. Specific error information is sent to the ULOG file.

- `tmwsdlgen` is used to generate the WSDL document file. If the validation fails, `tmwsdlgen` prints the error information to the console.

Dynamically Loading the Configuration

You can dynamically reload the BEA SALT configuration file by using the `wsadmin` command. For more information, see the BEA SALT Reference Guide.

Dynamically Reloading the SALT Configuration File

At runtime, the GWWS server can dynamically reload the configuration file. You can also download the WSDL document file from the GWWS server (which is based on the latest SALT configuration file).

Use the following command with `wsadmin` to reload the configuration:

```
configreload(creload) –i InstanceID1
```

This sub command is used to trigger configuration runtime reloading for the specified GWWS process. Option `-i` *must* be specified to indicate one GWWS server for reloading.

Troubleshooting Reloading Configurations Dynamically

Possible reasons for configuration reload failure are:

- Maximum configuration instances used concurrently in the current system. The limitation for normal reloading is two instances.
- The current system is busy with another reload request. Only one reload request is allowed at a time.
- The reloaded configuration file contains serious grammar errors and cannot be loaded as a valid configuration instance.
- System error encountered while reloading, for example, a memory error or file IO error.

Generating the WSDL Document

A Web service is usually defined using a WSDL document and made available via SOAP. WSDL, Web Service Descriptive Language, is an XML format used to describe network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

Before generating a WSDL document for the Web client to access services, be sure you complete the following tasks:

- Create a BEA SALT configuration file.
- Validate the BEA SALT configuration file.
- Configure the GWWS gateway as a Tuxedo server by updating the `*SERVERS` section of the UBBCONFIG file.
- Verify that all Tuxedo service information is available through the Tuxedo Service Metadata Repository.

When the GWWS server is booted, it loads the specified SALT configuration file, which includes: reading the configuration XML file, validating the XML file, loading corresponding Tuxedo service contract information from Tuxedo Service Metadata Repository, and loading WS-ReliableMessaging policy definition file. The GWWS server can also dynamically reload the BEA SALT configuration file at runtime.

The GWWS process also automatically generates a WSDL document to reflect the latest configuration information so that it can be downloaded from the GWWS server via HTTP GET method. To generate a WSDL document file, use `tmwsdlgen`.

For more information, see `tmwsdlgen`.

For information about WSDL mapping rules for Tuxedo buffer types, see “Data Mapping and Conversions” on page 3-1.

Viewing the WSDL Document

The GWWS gateway server, maintains the most recent auto-generated WSDL documents. The GWWS server supports a specific `HTTP GET` request from any HTTP client to download WSDL documents. The GWWS server can accept the SOAP version and encoding style indication for different formatted WSDL documents.

You can download the latest WSDL document using the following URL:

```
“http(s)://<host>:<port>/ wsdl[?[SOAPversion=<1.1 | 1.2> ] [&encstyle=<doc
| rpc > ][&mappolicy=<pack|raw>][&toolkit=<wls|axis>]]”
```

SOAPversion=<1.1|1.2>

Specifies the SOAP version for the generated WSDL document. If a version is not specified, the default is value `1.1`.

encstyle=<doc|rpc>
Specifies the encoding style for the generated WSDL document. If a style is not specified, the default value is `doc`.

mappolicy=<pack|raw>
Specifies the data mapping policies for certain Tuxedo Typed buffers for the generated WSDL document. Currently, this option impacts CARRAY typed buffers only. If the argument is not specified, `pack` is used as the default value.

toolkit=<wls|axis>
Use this argument only if you have previously defined `mappolicy=raw`. Specify the client toolkit used so that the proper WSDL document description for a CARRAY typed buffer MIME attachment is generated. BEA SALT supports WebLogic Server and Axis for SOAP with Attachments. The default value is wls.

Sample WSDL Document

Using the following sample BEA SALT configuration file, a corresponding WSDL document can be generated.

```
<Configuration xmlns=”http://www.bea.com/Tuxedo/Salt/200606” >
  <Servicelist id=”sample”>
     <Service name=”TOUPPER” />
  </Servicelist>
  <Policy />
  <System />
  <WSGateway>
     <GWInstance id=”GW1”>
        <HTTP address=”//webservice.com.abc:8080” />
     </GWInstance>
  </WSGateway>
</Configuration>
```

The following WSDL document is generated when the previous SALT configuration is used.

Listing 2-5 Sample WSDL Document

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:tuxtype="urn:pack.sample_typedef.salt11"
xmlns:tns="urn:sample.wsdl"
xmlns:soap11="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd" xmlns:wsrp="http://schemas.xmlsoap.org/rp/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="urn:sample.wsdl">
       <wsdl:documentation>Generated from conf.xml at 05-22-2006
14:27:26:773</wsdl:documentation>
       <wsdl:types>
              <xsd:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
targetNamespace="urn:pack.sample_typedef.salt11">
                     <xsd:element name="TOUPPER">
                            <xsd:complexType>
                                   <xsd:sequence>
                                          <xsd:element name="inbuf"
type="xsd:string"></xsd:element>
                                   </xsd:sequence>
                            </xsd:complexType>
                     </xsd:element>
                     <xsd:element name="TOUPPERResponse">
                            <xsd:complexType>
                                   <xsd:sequence>
                                          <xsd:element name="outbuf"
type="xsd:string"></xsd:element>
                                   </xsd:sequence>
                            </xsd:complexType>
                     </xsd:element>
              </xsd:schema>
       </wsdl:types>
       <wsdl:message name="TOUPPERInput">
              <wsdl:part element="tuxtype:TOUPPER"
name="parameters"></wsdl:part>
       </wsdl:message>
       <wsdl:message name="TOUPPEROutput">
              <wsdl:part element="tuxtype:TOUPPERResponse"
name="parameters"></wsdl:part>
       </wsdl:message>
       <wsdl:portType name="sample_PortType">
              <wsdl:operation name="TOUPPER">
                     <wsdl:input message="tns:TOUPPERInput"></wsdl:input>
                     <wsdl:output message="tns:TOUPPEROutput"></wsdl:output>
              </wsdl:operation>
       </wsdl:portType>
       <wsdl:binding name="sample_Binding" type="tns:sample_PortType">
              <soap11:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"></soap11:binding>
              <wsdl:operation name="TOUPPER">
```

```
                    <soap11:operation soapAction="TOUPPER"
style="document"></soap11:operation>
                    <wsdl:input>
                           <soap11:body use="literal"></soap11:body>
                    </wsdl:input>
                    <wsdl:output>
                           <soap11:body use="literal"></soap11:body>
                    </wsdl:output>
             </wsdl:operation>
       </wsdl:binding>
       <wsdl:service name="TuxedoWebService">
             <wsdl:port binding="tns:sample_Binding"
name="sample_GW1_HTTPPort">
                    <soap11:address
location="http://webservice.com.abc:8080/sample"></soap11:address>
             </wsdl:port>
       </wsdl:service>
</wsdl:definitions>
```

See Also

- UBBCONFIG(5)
- GWWS, `tmwsdlgen`, `wsadmin`
- Managing the Tuxedo Service Metadata Repository
- Troubleshooting

Data Mapping and Conversions

This section contains the following topics:

Overview

Each Tuxedo buffer type is described using an XML Schema in the generated WSDL document. This means that Tuxedo service request/response data is represented in regular XML format. The XML format input data is automatically converted into Tuxedo typed buffers according to the corresponding buffer type schema definitions.

The converted typed buffers are used as the input of the Tuxedo service. Any typed buffer returned by the Tuxedo service is converted into XML format and returned to the Web service client in SOAP response message.

Converting Tuxedo Buffers to/from XML

The GWWS server automatically converts SOAP message into Tuxedo buffer types and Tuxedo buffer types into XML SOAP messages. BEA SALT provides a set of rules for describing Tuxedo typed buffers in an XML document. These rules are exported as XML Schema definitions in

SALT WSDL documents. This simplifies buffer conversion and does not require previous knowledge about Tuxedo buffer types.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
`STRING`	Tuxedo STRING typed buffers are used to store character strings that terminate with a NULL character. Tuxedo STRING typed buffers are self-describing buffer.	xsd:string In the SOAP message, the XML element that encapsulates the actual string data, must be defined with xsd:string directly. **Notes:** • `STRING` data type can be specified with a max data length in the Tuxedo Service Metadata Repository. If defined in Tuxedo, the corresponding SOAP message also enforces this maximum. The GWWS server validates the actual message byte length against the definition in Tuxedo Service Metadata Repository. A SOAP fault message returns if the message byte length exceeds suported maximums. • The GWWS gateway process requires that the XML document is encoded with “UTF-8”. So the string data in the SOAP message can only be in “UTF-8” encoding.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
`CARRAY (Mapping with SOAP Message plus Attachments)`	Tuxedo CARRAY typed buffers store character arrays, any of which can be NULL. CARRAY buffers are used to handle data opaquely and are not self-describing.	The CARRAY buffer raw data is carried within a MIME multipart/related message, which is defined in the “SOAP Messages with Attachments”specification. The two data formats supported for MIME Content-Type attachments are: • `application/octet-stream` – Use for Apache Axis • `text/xml` – Use for BEA WebLogic Server The format depends on which Web service client side toolkit is used. **Note:** The SOAP with Attachment rule is only interoperable with BEA WebLogic Server and Apache Axis. For more information, see “Generating the WSDL Document” on page 2-25. **Notes:** CARRAY data type can be specified with a max byte length. If Tuxedo has such a definition, the corresponding SOAP message will also be enforced with this limitation. The GWWS server validates the actual message byte length against the definition in Tuxedo Service Metadata Repository.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
`CARRAY (Mapping with base64Binary)`	Tuxedo CARRAY typed buffers store character arrays, any of which can be NULL. CARRAY buffers are used to handle data opaquely and are not self-describing.	`xsd:base64Binary` The CARRAY data bytes must be encoded with `base64Binary` before it can be embedded into a SOAP message. Using `base64Binary` encoding with this opaque data stream saves the original data and makes the embedded data well-formed and readable. In the SOAP message, the XML element that encapsulates the actual CARRAY data, must be defined with `xsd:base64Binary` directly. For more information, see "Generating the WSDL Document" on page 2-25. **Note:** If your Tuxedo service uses `tpxmltofml`, the `<carray>` tag value must be a *hex string*. **Notes:** CARRAY data type can be specified with a max byte length. If defined in Tuxedo, the corresponding SOAP message is be enforced with this limitation. The GWWS server validates the actual message byte length against the definition in Tuxedo Service Metadata Repository.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
`MBSTRING`	Tuxedo MBSTRING Typed Buffers are used for multibyte character arrays. Tuxedo MBSTRING buffers consist of the following three elements: • Code-set character encoding • Data length • Character array of the encoding.	`xsd:string` The XML Schema built-in type, `xsd:string`, represents the corresponding type for buffer data stored in a SOAP message. The GWWS server only accepts "UTF-8" encoded XML documents. If Web service client wants to access Tuxedo services with MBSTRING buffer, the mbstring payload must be represented as "UTF-8" encoding in the SOAP request message. **Note:** The GWWS server transparently passes the "UTF-8" character set string into the Tuxedo service with MBSTRING Typed buffer format, and then the actual Tuxedo services handles the UTF-8 string. For any Tuxedo response MBSTRING Typed buffer (with any encoding character set), The GWWS server automatically transforms the string into "UTF-8" encoding and sends it back to the Web service client.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
MBSTRING (cont.)		**Limitation:** Tuxedo MBSTRING data type can be specified with a max byte length in the Tuxedo Service Metadata Repository. if defined in Tuxedo, the corresponding SOAP message should also be enforced with this limitation. **Note:** The BEA SALT WSDL generator will not have xsd:maxLength restrictions in the generated WSDL document, but the GWWS server will validate the byte length according to the Tuxedo Service Metadata Repository definition.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
XML	Tuxedo XML typed buffers store XML documents.	`xsd:anyType` The XML Schema built-in type, xsd:anyType, is the corresponding type for XML documents stored in a SOAP message which allows you to encapsulate any well-formed XML data within the SOAP message. **Limitation:** The GWWS server validates that the actual XML data is well-formed, but will not do any other enforcement validation, such as schema validation. Only a single root XML buffer is allowed to be stored in the SOAP body; the GWWS server checks for this. Be sure the actual XML data is encoded using the "UTF-8" character set. Any original XML document prolog information cannot be carried within the SOAP message. XML data type can specify a max byte data length. If defined in Tuxedo, the corresponding SOAP message also must enforce this limitation. **Note:** The BEA SALT WSDL generator will not have xsd:maxLength restrictions in the generated WSDL document, but the GWWS server will validate the byte length according to the Tuxedo Service Metadata Repository definition.
X_C_TYPE	X_C_TYPE buffer types are equivalent to VIEW buffer types.	See `VIEW/VIEW32`

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
`X_COMMON`	X_COMMON buffer types are equivalent to VIEW buffer type, but are used for compatibility between COBOL and C programs. Field types should be limited to short, long, and string	See `VIEW/VIEW32`
`X_OCTET`	X_OCTET buffer types are equivalent to CARRAY buffer types	See CARRAY `xsd:base64Binary`

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
`VIEW/VIEW32`	Tuxedo VIEW and VIEW32 typed buffers store C structures defined by Tuxedo applications. VIEW structures are defined by using VIEW definition files. A VIEW buffer type can define multiple fields. VIEW supports the following field types: • short • int • long • float • double • char • string • carray VIEW32 supports all the VIEW field types and mbstring.	Each VIEW or VIEW32 data type is defined as an XML Schema complex type. Each VIEW field should be one or more sub elements of the XML Schema complex type. The name of the sub element is the VIEW field name. The occurrence of the sub element depends on the count attribute of the VIEW field definition. The value of the sub element should be in the VIEW field data type corresponding XML Schema type. The the field types and the corresponding XML Schema type are listed as follows: • `short` maps to `xsd:short` • `int` maps to `xsd:int` • `long` maps to `xsd:long` • `float` maps to `xsd:float` • `double` maps to `xsd:double` • `char` (defined as byte in Tuxedo Service Metadata Repository definition) maps to `xsd:byte` • `char` (defined as char in Tuxedo Service Metadata Repository definition) maps to `xsd:string` (with restrictions maxlength=1) • `string` maps to `xsd:string` • `carray` maps to `xsd:base64Binary` • `mbstring` maps to `xsd:string`
`VIEW/VIEW32 (cont.)`		For limitations and considerations regarding mapping VIEW/VIEW32 buffers, refer to "VIEW/VIEW32 Considerations" on page 3-21.

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
FML/FML32	Tuxedo FML and FML32 typed buffers are proprietary BEA Tuxedo system self-describing buffers in which each data field carries its own identifier, an occurrence number, and possibly a length indicator. FML supports the following field types: • FLD_CHAR • FLD_SHORT • FLD_LONG • FLD_FLOAT • FLD_DOUBLE • FLD_STRING • FLD_CARRAY FML32 supports all the FML field types and FLD_PTR, FLD_MBSTRING, FLD_FML32, and FLD_VIEW32.	FML/FML32 buffers can only have basic data-dictionary-like definitions for each basic field data. A particular FML/FML32 buffer definition should be applied for each FML/FML32 buffer with a different type name. Each FML/FML32 field should be one or more sub -elements within the FML/FML32 buffer XML Schema type. The name of the sub element is the FML field name. The occurrence of the sub element depends on the count and requiredcount attribute of the FML/FML32 field definition. The e field types and the corresponding XML Schema type are listed below: • `short` maps to `xsd:short` • `int` maps to `xsd:int` • `long` maps to `xsd:long` • `float` maps to `xsd:float` • `double` maps to `xsd:double` • `char` (defined as byte in Tuxedo Service Metadata Repository definition) maps to `xsd:byte` • `char` (defined as char in Tuxedo Service Metadata Repository definition) maps to `xsd:string` • `string` maps to `xsd:string` • `carray` maps to `xsd:base64Binary` • `mbstring` maps to `xsd:string`

Table 3-1 Tuxedo Buffer Mapping to XML Schema

Tuxedo Buffer Type	Description	XML Schema Mapping for SOAP Message
FML/FML32 (cont.)		• view32 maps to tuxtype:view <viewname> • fml32 maps to tuxtype:fml32 <svcname>_p<SeqNum> To avoid multiple embedded FML32 buffers in an FML32 buffer, a unique sequence number (<SeqNum>) is used to distinguish the embedded FML32 buffers. **Note:** ptr is not supported. For limitations and considerations regarding mapping FML/FML32 buffers, refer to "FML/FML32 Considerations" on page 3-25.

Tuxedo STRING Typed Buffers

Tuxedo STRING typed buffer are used to store character strings that terminate with a NULL character. Tuxedo STRING typed buffers are self-describing buffer.

The following example depicts the TOUPPER Tuxedo service, which accepts a STRING typed buffer. The SOAP message is as follows:

```
<?xml … encoding="UTF-8" ?>
   ……
   <SOAP:body>
      <m:TOUPPER xmlns:m="urn:……">
           <inbuf>abcdefg</inbuf>
      </m:TOUPPER>
   </SOAP:body>
```

The XML Schema for <inbuf> is:

```
<xsd:element  name="inbuf" type="xsd:string" />
```

Tuxedo CARRAY Typed Buffers

Tuxedo CARRAY typed buffers are used to store character arrays, any of which can be NULL. They are used to handle data opaquely and are not self-describing. Tuxedo CARRAY typed buffers can map to `xsd:base64Binary` or MIME attachments. The default is `xsd:base64Binary`.

Mapping Example Using base64Binary

Listing 3-1 shows the SOAP message for the `TOUPPER` Tuxedo service, which accepts a `CARRAY` typed buffer, using `base64Binary` mapping.

Listing 3-1 Soap Message for a CARRAY Typed Buffer Using base64Binary Mapping

```
<SOAP:body>
       <m:TOUPPER xmlns:m=”urn:......”>
              <inbuf>QWxhZGRpbjpvcGVuIHNlc2FtZQ==</inbuf>
       </m:TOUPPER>
</SOAP:body>
```

The XML Schema for `<inbuf>` is:

```
<xsd:element  name=”inbuf” type=”xsd:base64Binary” />
```

Mapping Example Using MIME Attachment

Listing 3-2 shows the SOAP message for the `TOUPPER` Tuxedo service, which accepts a `CARRAY` typed buffer, as a MIME attachment.

Listing 3-2 Soap Message for a CARRAY Typed Buffer Using MIME Attachment

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
    start="<claim061400a.xml@example.com>"
Content-Description: This is the optional message description.

--MIME_boundary
```

```
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@ example.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
..
<m:TOUPPER xmlns:m=”urn:…”>
<inbuf href="cid:claim061400a.carray@example.com"/>
</m:TOUPPER>
..
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: text/xml
Content-Transfer-Encoding: binary
Content-ID: <claim061400a. carray @example.com>

...binary carray data…
--MIME_boundary--
The WSDL for carray typed buffer will look like the following:
<wsdl:definitions …>
<wsdl:types …>
       <xsd:schema …>
              <xsd:element name=”inbuf” type=”xsd:base64Binary” />
       </xsd:schema>
</wsdl:types>

……

<wsdl:binding …>
    <wsdl:operation name=”TOUPPER”>
           <soap:operation …>
           <input>
                 <mime:multipartRelated>
```

```
                    <mime:part>
                         <soap:body parts="…" use="…"/>
                    </mime:part>
                    <mime:part>
                         <mime:content part="…" type="text/xml"/>
                    </mime:part>
               </mime:multipartRelated>
          </input
          ……
    </wsdl:operation>
</wsdl:binding>

</wsdl:definitions>
```

Tuxedo MBSTRING Typed Buffers

Tuxedo MBSTRING Typed Buffers are used for multibyte character arrays. Tuxedo MBSTRING buffers consist of the following three elements: code-set character encoding, data length, character array encoding.

Note: You cannot embed multibyte characters with non "UTF-8" code sets into the SOAP message directly.

Figure 3-1 shows the SOAP message for the `MBSERVICE` Tuxedo service, which accepts an `MBSTRING` typed buffer.

Figure 3-1 SOAP Message for an MBSTRING Buffer

```
<?xml encoding="UTF-8" ?>
  <SOAP:body>
      <m: MBSERVICE xmlns:m="http://.....">
           <inbuf>こんにちは</inbuf>
      </m: MBSERVICE >
  </SOAP:body>
```

The XML Schema for `<inbuf>` is:

```
<xsd:element  name="inbuf" type="xsd:string" />
```

WARNING: BEA SALT converts the Japanese character "—" (EUC-JP 0xa1bd, Shift-JIS 0x815c) into UTF-16 0x2015.

If you use another character set conversion engine, the EUC-JP or Shift-JIS multibyte output for this character may be different. For example, the Java il8n character conversion engine, converts this symbol to UTF-16 0x2014. The result is the also same when converting to UTF-8, which is the BEA SALT default

If you use another character conversion engine and Japanese "—" is included in MBSTRING, TUXEDO server-side MBSTRING auto-conversion cannot convert it back into Shift-JIS or EUC-JP.

Tuxedo XML Typed Buffers

Tuxedo XML typed buffers store XML documents.

Listing 3-3 shows the Stock Quote XML document.

Listing 3-4 shows the SOAP message for the `STOCKINQ` Tuxedo service, which accepts an `XML` typed buffer.

Listing 3-3 Stock Quote XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "Stock Quotes". -->
<stockquotes>
  <stock_quote>
     <symbol>BEAS</symbol>
     <when>
       <date>01/27/2001</date>
       <time>3:40PM</time>
     </when>
     <change>+2.1875</change>
     <volume>7050200</volume>
  </stock_quote>
</stockquotes>
```

Then part of the SOAP message will look like the following:

Listing 3-4 SOAP Message for an XML Buffer

```
<SOAP:body>
       <m: STOCKINQ xmlns:m=”urn:......”>
       <inbuf>
       <stockquotes>
              <stock_quote>
                     <symbol>BEAS</symbol>
                     <when>
                     <date>01/27/2001</date>
                     <time>3:40PM</time>
                     </when>
                     <change>+2.1875</change>
                     <volume>7050200</volume>
              </stock_quote>
       </stockquotes>
       </inbuf>
       </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is:

```
<xsd:element  name=”inbuf” type=”xsd:anyType” />
```

Note: If a default namespace is contained in a Tuxedo XML typed buffer and returned to the GWWS server, the GWWS server converts the default namespace to a regular name. Each element is then prefixed with this name.

For example, If a Tuxedo service returns a buffer having a default namespace to the GWWS server as shown in Listing 3-5, the GWWS server converts the default namespace to a regular name as shown in Listing 3-6.

Listing 3-5 Default Namespace Before Sending to GWWS Server

```
<Configuration xmlns="http://www.bea.com/Tuxedo/Salt/200606">
       <Servicelist id="simpapp">
              <Service name="toupper"/>
       </Servicelist>
```

```
        <Policy/>
        <System/>
        <WSGateway>
          <GWInstance id="GWWS1">
               <HTTP address="//myhost:8080"/>
          </GWInstance>
        </WSGateway>
</Configuration>
```

Listing 3-6 GWWS Server Converts Default Namespace to Regular Name

```
        <dom0:Configuration
        xmlns:dom0="http://www.bea.com/Tuxedo/Salt/200606">
               <dom0:Servicelist dom0:id="simpapp">
                      <dom0:Service dom0:name="toupper"/>
               </dom0:Servicelist>
               <dom0:Policy></<dom0:Policy>
               <dom0:System></<dom0:System>
               <dom0:WSGateway>
                 <dom0:GWInstance dom0:id="GWWS1">
                      <dom0:HTTP dom0:address="//myhost:8080"/>
                 </dom0:GWInstance>
               </dom0:WSGateway>
        </dom0:Configuration>
```

Tuxedo VIEW/VIEW32 Typed Buffers

Tuxedo VIEW and VIEW32 typed buffers are used to store C structures defined by Tuxedo applications. You must define the VIEW structure with the VIEW definition files. A VIEW buffer type can define multiple fields.

Listing 3-7 shows the `MYVIEW` VIEW definition file.

Listing 3-8 shows the SOAP message for the MYVIEW Tuxedo service, which accepts a VIEW typed buffer.

Listing 3-7 VIEW Definition File for MYVIEW Service

```
VIEW MYVIEW
#type     cname       fbname       count       flag       size       null
float     float1       -            1           -          -          0.0
double    double1      -            1           -          -          0.0
long      long1        -            3           -          -          0
string    string1      -            2           -          20         '\0'
END
```

Listing 3-8 SOAP Message for a VIEW Typed Buffer

```
<SOAP:body>
       <m: STOCKINQ xmlns:m=”http://......”>
              <inbuf>
                     <float1>12.5633</float1>
                     <double1>1.3522E+5</double1>
                     <long1>1000</long1>
                     <long1>2000</long1>
                     <long1>3000</long1>
                     <string1>abcd</string1>
                     <string1>ubook</string1>
              </inbuf>
       </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is shown in Listing 3-9.

Listing 3-9 XML Schema for a VIEW Typed Buffer

```
<xsd:complexType name=" view_MYVIEW">
  <xsd:sequence>
       <xsd:element name="float1" type="xsd:float" />
       <xsd:xsd:element name="double1" type="xsd:double" />
       <xsd:element name="long1" type="xsd:long" minOccurs="3" />
       <xsd:element name="string1" type="xsd:string minOccurs="3" />
  </xsd:sequence>
</xsd: complexType >
<xsd:element  name="inbuf" type="tuxtype:view_MYVIEW" />
```

VIEW/VIEW32 Considerations

The following considerations apply when converting Tuxedo VIEW/VIEW32 buffers to and from XML.

- You must create an environment for converting XML to and from VIEW/VIEW32. This includes setting up a VIEW directory and system VIEW definition files. These definitions are automatically loaded by the GWWS server.
- The GWWS server provides strong consistency checking between the Tuxedo Service Metadata Repository VIEW/VIEW32 parameter definition and the VIEW/VIEW32 definition file at start up.

  If an inconsistency is found, the GWWS server cannot start. Inconsistency messages are printed in the ULOG file.
- `tmwsdlgen` also provides strong consistency checking between the Tuxedo Service Metadata Repository VIEW/VIEW32 parameter definition and the VIEW/VIEW32 definition file at start up. If an inconsistency is found, the GWWS server will not start. Inconsistency messages are printed in the ULOG file.

  If the VIEW definition file cannot be loaded, `tmwsdlgen` attempts to use the Tuxedo Service Metadata Repository definitions to compose the WSDL document.
- Because `dec_t` is not supported, if you define VIEW fields with type `dec_t`, the service cannot be exported as a Web service and an error message is generated when the BEA SALT configuration file is loading.

- Although the Tuxedo Service Metadata Repository may define a size attribute for “string/mbstring” typed parameters, which represents the maximum byte length that is allowed in the Tuxedo typed buffer, BEA SALT does not expose such restriction in the generated WSDL document.
- When a VIEW32 embedded MBString buffer is requested and returned to the GWWS server, the GWWS miscalculates the required MBString length and reports that the input string exceeds the VIEW32 maxlength. This is because the header is included in the transfer encoding information. You must include the header size when defining the VIEW32 field length.
- The Tuxedo primary data type “long” is indefinite between 32-bit and 64-bit scope, depending on the platform. However, the corresponding `xsd:long` schema type is used to describe 64-bit numeric values.

  If the GWWS server runs in 32-bit mode, and the Web service client sends `xsd:long` typed data that exceeds the 32-bit value range, you may get a SOAP fault.

Tuxedo FML/FML32 Typed Buffers

Tuxedo FML and FML32 typed buffer are proprietary BEA Tuxedo system self-describing buffers in which each data field carries its own identifier, an occurrence number, and possibly a length indicator.

FML Data Mapping Example

Listing 3-10 shows the SOAP message for the `TRANSFER` Tuxedo service, which accepts an `FML` typed buffer.

The request fields for service `LOGIN` are:

```
ACCOUNT_ID       1      long            /* 2 occurrences, The withdrawal
account is 1st, and the deposit account is 2nd  */
AMOUNT              2       float           /* The amount to transfer */
```

Part of the SOAP message is as follows:

Listing 3-10 SOAP Message for an FML Typed Buffer

```
<SOAP:body>
       <m:TRANSFER xmlns:m=”urn:......”>
              <inbuf>
```

```
                    <ACCOUNT_ID>40069901</ACCOUNT_ID>
                    <ACCOUNT_ID>40069901</ACCOUNT_ID>
                    <AMOUNT>200.15</AMOUNT>
               </inbuf>
       </m:TRANSFER >
</SOAP:body>
```

The XML Schema for <inbuf> is shown in Listing 3-11.

Listing 3-11 XML Schema for an FML Typed Buffer

```
<xsd:complexType name=" fml_TRANSFER_In">
  <xsd:sequence>
       <xsd:element name="ACCOUNT_ID" type="xsd:long" minOccurs="2"/>
       <xsd:element name=" AMOUNT" type="xsd:float" />
  </xsd:sequence>
</xsd: complexType >
<xsd:element  name="inbuf" type="tuxtype: fml_TRANSFER_In" />
```

FML32 Data Mapping Example

Listing 3-12 shows the SOAP message for the TRANSFER Tuxedo service, which accepts an FML32 typed buffer.

The request fields for service LOGIN are:

```
CUST_INFO           1       fml32         /* 2 occurrences, The withdrawal
customer is 1st, and the deposit customer is 2nd  */
ACCOUNT_INFO   2         fml32          /* 2 occurrences, The withdrawal
account is 1st, and the deposit account is 2nd  */
AMOUNT               3        float            /* The amount to transfer */
```

Each embedded CUST_INFO includes the following fields:

```
CUST_NAME       10       string
CUST_ADDRESS    11       carray
CUST_PHONE      12       long
```

Each embedded ACCOUNT_INFO includes the following fields:

```
ACCOUNT_ID        20        long
ACCOUNT_PW        21        carray
```

Part of the SOAP message will look as follows:

Listing 3-12 SOAP Message for Service with FML32 Buffer

```
<SOAP:body>
   <m:STOCKINQ xmlns:m=”urn:......”>
       <inbuf>
              <CUST_INFO>
                     <CUST_NAME>John</CUST_NAME>
                     <CUST_ADDRESS>Building 15</CUST_ADDRESS>
                     <CUST_PHONE>1321</CUST_PHONE>
              </CUST_INFO>
              <CUST_INFO>
                     <CUST_NAME>Tom</CUST_NAME>
                     <CUST_ADDRESS>Building 11</CUST_ADDRESS>
                     <CUST_PHONE>1521</CUST_PHONE>
              </CUST_INFO>
              <ACCOUNT_INFO>
                     <ACCOUNT_ID>40069901</ACCOUNT_ID>
                     <ACCOUNT_PW>abc</ACCOUNT_PW>
              </ACCOUNT_INFO>
              <ACCOUNT_INFO>
                     <ACCOUNT_ID>40069901</ACCOUNT_ID>
                     <ACCOUNT_PW>zyx</ACCOUNT_PW>
              </ACCOUNT_INFO>

              <AMOUNT>200.15</AMOUNT>
       </inbuf>
   </m: STOCKINQ >
</SOAP:body>
```

The XML Schema for <inbuf> is shown in Listing 3-13.

Listing 3-13 XML Schema for an FML32 Buffer

```
<xsd:complexType name=”fml32_TRANSFER_In”>
  <xsd:sequence>
     <xsd:element name=”CUST_INFO” type=”tuxtype:fml32_TRANSFER_p1”
minOccurs=”2”/>
     <xsd:element name=”ACCOUNT_INFO” type=”tuxtype:fml32_TRANSFER_p2”
minOccurs=”2”/>
     <xsd:element name=”AMOUNT” type=”xsd:float” />
  /xsd:sequence>
</xsd:complexType >

<xsd:complexType name=”fml32_TRANSFER_p1”>
     <xsd:element name=”CUST_NAME” type=”xsd:string” />
     <xsd:element name=”CUST_ADDRESS” type=”xsd:base64Binary” />
     <xsd:element name=”CUST_PHONE” type=”xsd:long” />
</xsd:complexType>

<xsd:complexType name=”fml32_TRANSFER_p2”>
     <xsd:element name=”ACCOUNT_ID” type=”xsd:long” />
     <xsd:element name=”ACCOUNT_PW” type=”xsd:base64Binary” />
</xsd:complexType>

<xsd:element  name=”inbuf” type=”tuxtype: fml32_TRANSFER_In” />
```

FML/FML32 Considerations

The following considerations apply to converting Tuxedo FML/FML32 buffers to and from XML.

- You must have an environment for converting XML to and from FML/FML32. This includes an FML field table file directory and system FML field definition files. These definitions are automatically loaded by GWWS. FML typed buffer can be handled only if user sets up the environment correctly.
- FML32 Field type `FLD_PTR` is not supported.

- The GWWS server provides strong consistency checking between the Tuxedo Service Metadata Repository FML/FML32 parameter definition and FML/FML32 definition file during start up.

  If any FML/32 field cannot be found according to the environment setting, or the field data type definition in the field table is different from the parameter data type definition in the Tuxedo Service Metadata Repository, the GWWS cannot start. Inconsistency messages are printed in the ULOG file.

- the `tmwsdlgen` command checks for consistency between the Tuxedo Service Metadata Repository FML/FML32 parameter definition and FML/FML32 definition file, but will issue a warning and allow inconsistencies.

  If any FML/32 field cannot be found according to the environment setting, or the field data type definition in the field table is different from the parameter data type definition in Tuxedo Service Metadata Repository, `tmwsdlgen` attempts to use Tuxedo Service Metadata Repository definitions to compose the WSDL document.

- Although the Tuxedo Service Metadata Repository may define a size attribute for "string/mbstring" typed parameters, which represents the maximum byte length that is allowed in the Tuxedo typed buffer, BEA SALT does not expose such restriction in the generated WSDL document.

- Tuxedo primary data type "long" is indefinite between 32-bit and 64-bit scope according to different platforms. But the corresponding `xsd:long` schema type is used to describe 64-bit numeric value. You should get a SOAP fault in the following case: The GWWS runs with the 32-bit mode, and Web service client sends a `xsd:long` typed data which exceeds the 32-bit value range.

Tuxedo X_C_TYPE Typed Buffers

Tuxedo X_C_TYPE typed buffers are equivalent, and have similar WSDL format to Tuxedo VIEW typed buffers.They are transparent for SOAP clients. However, even though usage is similar to the Tuxedo VIEW buffer type, SALT administrators must configure the Tuxedo Service Metadata Repository for any particular Tuxedo service which uses this buffer type.

Note: All View related considerations also take effect for X_C_TYPE typed buffer.

Tuxedo X_COMMON Typed Buffers

Tuxedo X_COMMON typed buffers are equivalent to Tuxedo VIEW typed buffers. However, they are used for compatibility between COBOL and C programs. Field types should be limited to short, long, and string.

Tuxedo X_OCTET Typed Buffers

Tuxedo X_OCTET typed buffers are equivalent to CARRAY.

Note: Tuxedo X_OCTET typed buffers can only map to `xsd:base64Binary` type. SALT 1.1 does not support MIME attachment binding for Tuxedo X_OCTET typed buffers.

Custom Typed Buffers

BEA SALT provides a plug-in mechanism to support custom typed buffers. You can validate the SOAP message against your own XML Schema definition, allocate custom typed buffers, and parse data into the buffers and other operations.

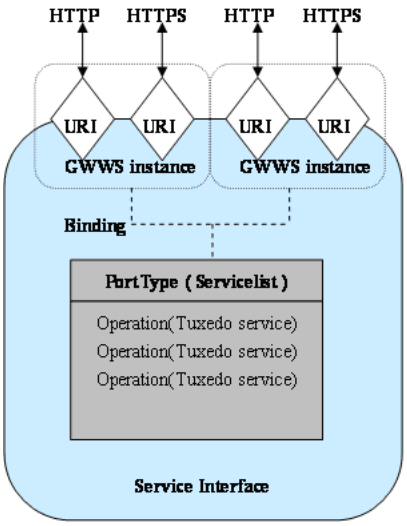
XML Schema built-in type `xsd:anyType` is the corresponding type for XML documents stored in a SOAP message. While using custom typed buffers, you should define and represent the actual data into an XML format and transfer between Web service client and Tuxedo Web service stack. Similar to the XML typed buffers, only a single root XML buffer is allowed to be stored in the SOAP body. The GWWS checks this for consistency.

For more plug-in information, see "Introduction to Using Plug-ins with BEA SALT" on page 5-1.

WSDL Mapping Rules

In order to export Tuxedo services and Web services, Tuxedo service contract information is needed so that it can be converted into a WDSL document. SALT leverages the Tuxedo Service Metadata Repository to access Tuxedo service information.

Figure 3-2 WSDL Component Model



WSDL Component Model

Converting configuration information into WSDL specification compliant information requires mapping rules. Table 3-2 provides the mapping rules for a generated WSDL document.

Table 3-2 Basic WSDL Mapping Rules

WSDL Definition	Description
`/wsdl:definitions/wsdl:types`	Uses XML Schema as type system
`/wsdl:definitions/wsdl:message`	Each instance maps to a specific Tuxedo service input buffer or output buffer

Table 3-2 Basic WSDL Mapping Rules

WSDL Definition	Description
`/wsdl:definitions/wsdl:portType`	Only one instance in the generated WSDL document. All Tuxedo services defined in the certain configuration instance are grouped in one WSDL portType object. Use `<wsdl:portType>` to define a set of abstract operations. To associate a Tuxedo service with a `<wsdl:portType>` definition, use the following rules: • `/wsdl:portType/@name` The `name` attribute value uses the following syntax: `<Servicelist_id>_PortType` • `/wsdl:portType/wsdl:operation/@name` The string value of attribute name should match the associated service name defined in the SALT configuration file and Tuxedo Service Metadata Repository.
`/wsdl:definitions/wsdl:portType/wsdl:operation`	Each instance maps to a specific Tuxedo service
`/wsdl:definitions/wsdl:binding`	Only one instance in the generated WSDL document. It indicates that, only one concrete SOAP protocol, either SOAP 1.1 binding extension or SOAP 1.2 binding extension, is provided for the WSDL binding in the generated WSDL document.
`/wsdl:definitions/wsdl:service`	Only one instance in the generated WSDL.

Table 3-2 Basic WSDL Mapping Rules

WSDL Definition	Description
`/wsdl:definitions/wsdl:port`	One or more instances in the generated WSDL document. It depends on the GWWS instance number used for failover.
`WSDL target Namespace`	The target Namespace of the generated WSDL document is composed using the following syntax: `urn:<Servicelist_ID>.wsdl` The first GWWS instance network parameters are used. Use the following Namespace conventions for each prefix: • `xsd` uses `http://www.w3.org/2001/XMLSchema` • `wsdl` uses `http://schemas.xmlsoap.org/wsdl` • `soap11` uses `http://schemas.xmlsoap.org/wsdl/soap` • `soap12` uses `http://schemas.xmlsoap.org/wsdl/soap12` • `wsp` uses `http://schemas.xmlsoap.org/ws/2004/09/policy` • `wsrm` uses `http://schemas.xmlsoap.org/ws/2005/02/rm` • `beapolicy` uses `http://www.bea.com/wsrm/policy`

WS Policy Attachment Rules

The following rules apply for attaching WS Policy files to WSDL documents.

- Each WS policy file content must be excerpted completely and attached as child elements of `<wsdl:definitions>` with the top element as `<wsp:Policy>`.

- Each policy expression defines the `wsu:id` attribute with a unique ID name.
- The `<wsp:UsingPolicy>` element is mandatory and must be a child element of `<wsdl:definitions>`. The format of `<wsp:UsingPolicy>` must be a fixed format as follows:

  ```
  <wsp:UsingPolicy wsdl:required="true" />
  ```

- The WS Policy is defined in the SALT configuration file and attached as an element in the WSDL document.

Policy Definition in SALT Configuration	Attached as Child Element
`/Configuration/Policy/RMPolicy`	`/wsdl:binding`

SOAP Message Exchange Pattern Mapping

BEA SALT supports the following mapping rules for each Tuxedo service type and SOAP Message Exchange Pattern (MEP).

Table 3-3 Tuxedo Service to SOAP Message Exchange Pattern Mapping

Tuxedo Service Type (defined in Tuxedo Service Metadata Repository)	SOAP Message Exchange Pattern (MEP)	Required Elements for <wsdl:operation>
`service/queue`	Request-Response	`<input>` `<output>`
`oneway`	One-way	`<input>`

Note: BEA SALT does not support the Tuxedo `queue` service type, even though it is supported in the Tuxedo Service Metadata Repository.

SOAP Message Encoding Support

BEA SALT supports two traditional message encoding styles for SOAP messages:

- Document style
- RPC style.

Document Message Style

The generated WSDL document supports the abstract message definition in document message style. Because the document style can represent SOAP messages in various XML document structures, there are many ways to define this document structure.

The following rules apply to SALT-generated WSDL documents to create a fixed-structure message description.

/wsdl:message/@name

The message name string syntax is as follows:

```
<Tuxedo service name + "Input | Output">
```

Use the *Input* suffix when describing an input typed buffer. Use the *Output* suffix when describing an output typed buffer

/wsdl:message/wsdl:part

The following format rules apply for `<wsdl:part>`:

- Only one `<wsdl:part>` instance is associated with each Tuxedo typed buffer.
- The value of `name` attribute of `<wsdl:part>` is hard-coded as `parameters`.
- The `element` attribute of `<wsdl:part>` is specified.

  If the message indicates a Tuxedo service *input buffer*, the value is the same as the associated Tuxedo service name.

  If the message indicates a Tuxedo service *output buffer*, the value is constructed with the associated Tuxedo service name, plus a suffix string response.

  The schema definition for `/wsdl:message/wsdl:part/@element` is:

  – `element` is `<xsd:complexType>`. The type name for this complex type is not defined; rather it is described by the type structure. The type name is not defined because the type indicating a particular service buffer is not likely to be reused for other elements, so there is no need to define a type name for reference.

  – `<xsd:complexType>` includes only one `element` with a Tuxedo buffer type name, for example, `STRING` or `XML`.

  – The type name of the buffer type element complies with the rules described Table 3-1.

  – For `VIEW/VIEW32` or `FML/FML32` or a custom type buffers described parameters, the `#bufmappingtype` definition includes parameter-specifying elements.

– Elements representing buffer parameters use a parameter name as an element name, and the type for the element complies with the rules described Table 3-1.

The following code listing shows the non-normative grammar:

```
<xsd:schema …>
 <xsd:element name=”#svcname” > *
  <xsd:complexType>
   <xsd:sequence>
       <xsd:element name=”#buffertype” type=”#bufmappingtype” />
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name=”#bufmappingtype”> *
    <xsd:sequence>
     <xsd:element name=”#parametername” type=”#parammappingtype” /> *
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name=” #parammappingtype”> *
    ……
 </xsd:complexType>

</xsd:schema>
```

Document Message Style Example

This example in this section depicts the `transfer` service definitions in the SALT configuration file, the corresponding `transfer` service contract information from the Tuxedo Service Metadata Repository, the generated WSDL document, and the associated request and response messages for the `transfer` service.

Listing 3-14 shows an example of a section of the BEA SALT configuration file definition for the service.

Listing 3-14 Document Message Transfer Service Servicelist Section of the SALT Configuration File

```
<Configuration>
   <Servicelist id=”bank”>
       <Service name=”transfer” />
   </Servicelist>
…….
</Configuration>
```

The corresponding Tuxedo Service Metadata Repository transfer service contract information is shown in Listing 3-15.

Listing 3-15 Tuxedo Service Metadata Repository Transfer Service Contract Information

```
Service=transfer
inbuf=FML
outbuf=STRING

param=ACCOUNT_ID
type=long
access=in
count=2
requiredcount=2

param=SAMOUNT
type=string
access=in

param=result
type=STRING
access=out
```

BEA SALT generates a WSDL document. Listing 3-16 shows the part of the WSDL document that relates to the `transfer` service definition.

Listing 3-16 Transfer Service Definition in a WSDL Document

```
<wsdl:definition
       xmlns:tuxtype="urn:bank_typedef"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       ……>
 <wsdl:types>
  <xsd:schema targetNamespace="urn:bank_typedef" …… >
```

```
   <xsd:element name=”transfer”>
    <xsd:complexType>
     <xsd:sequence>
       <xsd:element name=”FML” type=”tuxtype:fml_transfer_In” />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:complexType name=”fml_transfer_In”>
    <xsd:sequence>
       <xsd:element name=”ACCOUNT_ID” type=”xsd:long” minOccurs=”2”
maxOccurs=”2” />
       <xsd:element name=”SAMOUNT” type=”xsd:string” />
    </xsd:sequence>
   </xsd:complexType>

   <xsd:element name=”transferResponse”>
    <xsd:complexType>
     <xsd:sequence>
       <xsd:element name=”STRING” type=”xsd:string” />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   …….
  </xsd:schema>
 </wsdl:types>

 <wsdl:message name=”transferInput”>
  <wsdl:part name=”parameters” element=”tuxtype:transfer” />
 </wsdl:message>
 <wsdl:message name=”transferOutput”>
  <wsdl:part name=”parameters” element=”tuxtype:transferResponse” />
 </wsdl:message>

 ……
</wsdl:definition>
```

Listing 3-17 shows the SOAP *request* message for the `transfer` service.

Listing 3-17 Sample SOAP1.1 Transfer Service Request Message

```
<soap:envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
    <soap:body>
     <transfer xmlns=" urn:bank_typedef">
      <FML>
       < ACCOUNT_ID >111222</ACCOUNT_ID >
       < ACCOUNT_ID >333444</ACCOUNT_ID >
       < SAMOUNT > 100.00 </ SAMOUNT >
      </FML>
     </transfer>
    </soap:body>
</soap:envelope>
```

Listing 3-18 shows the SOAP *response* message for the `transfer` service.

Listing 3-18 Sample SOAP1.1 Transfer Service Response Message

```
<soap:envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
    <soap:body>
     <transferResponse xmlns="urn:bank_typedef">
       <STRING>30010.50</STRING>
     </transferResponse>
    </soap:body>
</soap:envelope>
```

RPC Message Style

The generated WSDL document supports abstract message definitions in RPC message style. The following rules apply to SALT-generated WSDL documents for creating RPC style message definitions.

/wsdl:message/@name

The string of message name is constructed with the following format:

`<Tuxedo service name + “Input | Output”>`

Use the *Input* suffix when describing an input typed buffer. Use the *Output* suffix when describing an output typed buffer

/wsdl:message/wsdl:part

The following format rules apply for `<wsdl:part>`:

- Only one `<wsdl:part>` instance is associated with each Tuxedo input message. The value of the `name` attribute of `<wsdl:part>` is hard-coded as `inbuf`.
- Only one `<wsdl:part>` instance is associated with each Tuxedo output message. The value of the `name` attribute of `<wsdl:part>` is hard-coded as `outbuf`.
- The type name of the buffer type element complies with the rules described Table 3-1.
- For `VIEW/VIEW32` or `FML/FML32` or custom type buffers described parameters, the `#bufmappingtype` definition includes parameter-specifying elements.
- Elements representing buffer parameters use the parameter name as an element name.The type for the element complies with the rules described Table 3-1.

  The following code listing shows the non-normative grammar:

```
<xsd:schema …>
   <xsd:complexType name=”bufmappingtype”> *
     <xsd:sequence>
       <xsd:element name=”#parametername” type=”#parammappingtype” /> *
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name=” #parammappingtype”> *
    ……
   </xsd:complexType>
</xsd:schema>
```

RPC Message Style Example

This example in this section depicts the `transfer` service definitions in the SALT configuration file, the corresponding `transfer` service contract information from the Tuxedo Service Metadata Repository, the generated WSDL document, and the associated request and response messages for the `transfer` service.

Listing 3-19 shows an example of a section of the BEA SALT configuration file definition for the service.

Listing 3-19 RPC Message Transfer Service Servicelist Section of the SALT Configuration File

```
<Configuration>
   <Servicelist id="bank">
     <Service name="transfer" />
   </Servicelist>
   ……
</Configuration>
```

Listing 3-20 shows the corresponding transfer service contract information retrieved from Tuxedo Service Metadata Repository.

Listing 3-20 Tuxedo Service Metadata Repository Transfer Service Contract Information

```
Service=transfer
inbuf=FML
outbuf=STRING

param=ACCOUNT_ID
type=long
access=in
count=2
requiredcount=2

param=SAMOUNT
type=string
access=in

param=result
type=STRING
access=out
```

BEA SALT generates a WSDL document. Listing 3-21 shows the part of the WSDL document that relates to the `transfer` service definition.

Listing 3-21 RPC Message Transfer Service Definition in the WSDL Document

```
<wsdl:definition
       xmlns:tuxtype=”urn:bank_typedef”
       xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”
       xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
       ……>
  <wsdl:types>
   <xsd:schema targetNamespace=”urn:bank_typedef” …… >
    <xsd:complexType name=”fml_transfer_In”>
     <xsd:sequence>
        <xsd:element name=”ACCOUNT_ID” type=”xsd:long” minOccurs=”2”
maxOccurs=”2” />
       <xsd:element name=”SAMOUNT” type=”xsd:string” />
     </xsd:sequence>
    </xsd:complexType>
    ……
   </xsd:schema>
  <wsdl:types>

  <wsdl:message name=”transferInput”>
    <wsdl:part name=”inbuf” type=”tuxtype: fml_transfer_In” />
  </wsdl:message>
  <wsdl:message name=”transferOutput”>
    <wsdl:part name=”outbuf” type=”xsd:string” />
  </wsdl:message>

  ……
</wsdl:definition>
```

Listing 3-22 shows the SOAP request message for the `transfer` service.

Listing 3-22 Sample SOAP1.1 Transfer Service Request RPC Message

```
<soap:envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
   <soap:body>
     <transfer xmlns:ns="urn:bank_typedef" >
       <inbuf  xsi:type="ns:fml_transfer_In" >
          <ACCOUNT_ID xsi:type="xsd:long">111222</ACCOUNT_ID >
          <ACCOUNT_ID xsi:type="xsd:long">333444</ACCOUNT_ID >
          <SAMOUNT xsi:type="xsd:float"> 100.00 </SAMOUNT >
       </inbuf>
     </transfer>
   </soap:body>
</soap:envelope>
```

See Also

Managing Typed Buffers

CHAPTER 4

Monitoring and Tuning Web Services

This section contains the following topics:

Viewing the Current Configuration

To view the current SALT configuration, use the following `wsadmin` command argument:

```
configstats(cstat)  -i InstanceID
```

This argument displays the current status of the SALT configuration file for the given GWWS process. It outputs the following:

- The number of current configuration instances used for the GWWS process
- Each configuration instance displays the following information:
  - Elapsed time: (xx) days (xx) hours (xx) minutes, (xx) seconds the duration since a configuration instance is loaded / reloaded.
  - Number of clients (SOAP requests) that are currently using this configuration instance.
  - Longest time of attachment for this instance: (xx) days (xx) hours (xx) minutes (xx) seconds.

Viewing Runtime Statistics

To view GWWS process runtime statistics, use the following `wsadmin` command argument:

```
gwstats(gws) -i InstanceID
```

The `gwstats` argument displays runtime statistical information for GWWS processes. Option `-i` is required.

Note: The runtime statistics do not apply to ~~work for~~ messages in a sequence for WS-ReliableMessaging specification.

Table 4-1 shows the resulting displayed information.

Table 4-1 Statistics Displayed in Terce Mode

Statistic	Description
Request Response Done	Number of successfully completed requests in request-response message pattern. A completed request in request-response message pattern means a SOAP response is returned from the GWWS server to the Web Service Client.
Request Response Fail	Number of failed requests in request-response message pattern. A failed request in request-response message pattern means a SOAP Fault message is returned from the GWWS server to the Web Service Client.Possible failure reasons: • Invalid SOAP request • Tuxedo internal error • SOAP request rejected
Oneway Done	Number of successfully completed requests. A completed request means a SOAP request is successfully delivered to a Tuxedo service regardless of whether the service process is successful or not.
Oneway Fail	Number of failed requests in oneway message pattern. A failed request in oneway message pattern means a SOAP oneway request is not delivered to the Tuxedo service.

Tuning the GWWS Server

The GWWS server is a high performance gateway used between SOAP clients and the Tuxedo framework. It uses a thread-pool working model to improve performance in a multi-processor

server environment. The GWWS server also provides options to control runtime behavior by setting <WSGateway> element property values in the BEA SALT configuration file. The following topics list deployment considerations based on different scenarios. For more information, see Configuring BEA SALT in the *BEA SALT Administration Guide*.

Thread Pool Size Tuning

Property: `thread_pool_size`

The default thread pool size is 16, but in some cases this may not be enough to handle high volume loads. It is recommended to conduct a typical usage analysis in order to better estimate the proper size requirement. Usually, if the concurrent client number is large (for example, more than 500), it is suggested that you deploy the GWWS gateway on a server with at least a 4-way processor and set the thread pool size to 64.

Network Timeout Control

Property: `timeout`

BEA SALT provides a network timeout tuning parameter in the configuration file. The default timeout value is 300 seconds.The value can be adjusted to reduce timeout errors.

Max Content Length Control

Property: `max_content_length`

BEA SALT administrators may want to limit the buffer size sent from a client. SALT supports this by using a property value that can be set for particular GWWS instances. By default there is no limit.

Backlog Control

Property: `max_backlog`

BEA SALT defines the default backlog socket listen value to 20. On some systems, such as Windows, 20 may not meet the heavy load requirements. The client connection is rejected during TCP handshake.

The recommended value for Windows is based on the max concurrent TCP connections you may encounter. For example, if 80 is the peak point, you may configure the `max_backlog` property value to 60 in the SALT configuration file.

Note: The default backlog value is adequate for most systems. You do not need tune it unless you experience client connection problems during heavy loads.

WARNING: A large backlog value may increase *syn-blood* attack risk.

Tuxedo BLOCKTIME

A network receive timeout property is provided in the SALT configuration file. Web service applications are also impacted by the Tuxedo BLOCKTIME parameter. Blocktime accounting begins when a message is transformed from XML to a typed buffer and delivered to the Tuxedo framework.

If no reply is received for a particular Web service client within the BLOCKTIME time frame, the GWWS server sends a SOAP fault message to the client and terminates the connection. If the GWWS server receives a delayed reply, it drops this message because the client has been disconnected.

BLOCKTIME is defined in the UBBCONFIG file `*RESOURCE` section.

Boost Performance Using Multiple GWWS instances

If one GWWS instance is bottlenecked due to network congestion, low CPU resources and so on, multiple GWWS instances can be deployed within the same SALT configuration file on distributed Tuxedo nodes.

Note: Even though multiple GWWS instances can provide the same logic functionality, from a client perspective, they are different Web service end points with specific TCP ports and addresses.

See Also

- GWWS

CHAPTER 5

Introduction to Using Plug-ins with BEA SALT

This section contains the following topics:

Overview

A plug-in is a set of functions that are called when the GWWS server is running. BEA SALT provides a plug-in framework as a common interface for defining and implementing a plug-in. Plug-in implementation is carried out through a dynamic library that contains the actual function code and exports it. The implementation library can be loaded dynamically during GWWS server startup. The functions are registered as the implementation of the plug-in interface.

For more information see, Using Plug-ins with BEA SALT in BEA SALT *Programming Web Services*.

Implementing a Plug-in with SALT

To use a plug-in with BEA SALT, you must do the following:

1. Create the plug-in dynamic library that contains the function code. Follow the guidelines in this section for creating that plug-in.
2. Specify the plug-in using the `<Plug-in>` system parameter in the SALT configuration file.

For more information about specifying a plug-in, see “Configuring BEA SALT” on page 2-1.

3. Initiate the GWWS server to dynamically load the plug-in. After loading a plug-in implementation, the plug-in function call is directed to the library.

Defining a Plug-in

There are basic requirements for a plug-in. These requirements include:

- Plug-in ID

  The plug-in ID is an arbitrary string that is unique to the plug-in so that it differentiates it from another plug-in. BEA SALT 1.1 only support the `P_CUSTOM_TYPE` plug-in ID.

- Plug-in name

  The plug-in name differentiates an implementation from other implementations of the same plug-in. The plug-in name *must* match the custom type buffer name defined in the Tuxedo Service Metadata Repository.

- Initiating function

  An initializing function called `_ws_pi_init_ID_Name()`, in which `ID` and `Name` are the ID of plug-in interface and the name of the plug-in implementation.

  The function is called when being registered. The initializing uses the following syntax:

  ```
  int _ws_pi_init_ID_Name(char * params, void **priv_ptr);
  ```

  The function should allocate all necessary resources and setup the vtable for the plug-in. When the register function initializes, the vtable interface can be setup if necessary.

  The first parameter is a string that is passed to the dynamic library for initialization. The second parameter stores private data that the implementation may need.

- Exiting function

  The exiting function uses the following syntax:

  ```
  int _ws_pi_exit_ID_Name(void * priv);
  ```

  The exiting function is called when the implementation is unregistered and a function to setup the vtable interface is called:

  ```
  int _ws_pi_set_vtbl_ID_Name(void *vtbl);
  ```

See Also

- Using Plug-ins with BEA SALT in BEA SALT *Programming Web Services*

CHAPTER 6

Interoperability Considerations

The generated WSDL document allows other Web service toolkits to develop Web service clients that can access Tuxedo service using the GWWS server. The generated WSDL document *must* be able to publish to a UUDI server.

Table 6-1 lists BEA SALT supported toolkit interoperability.

Table 6-1 Interoperability Support

	WebLogic Server			Axis for Java 1.3	WebLogic Workshop 8.1	.Net 2.0 with WSE 3.0	.Net 1.1
	9.1	8.1	9.2				
SOAP 1.1/DOC	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SOAP 1.1/RPC	Yes	Yes	Yes	Yes	Yes	N/A	N/A
SOAP 1.2/DOC	N/A	N/A	N/A	Yes	N/A	Yes	N/A
SOAP 1.2/RPC	N/A	N/A	N/A	Yes	N/A	N/A	N/A
WS-ReliableMessaging	Yes	N/A	N/A	N/A	N/A	N/A	N/A
WS-Addressing	Yes	N/A	N/A	N/A	N/A	N/A	N/A

APPENDIX A

Troubleshooting

This section contains the following topics:

GWWS Startup Failure

If the GWWS server fails to start, check the following:

- Tuxedo service contract configuration

  Check the Tuxedo service contract configuration in the Tuxedo Service Metadata Repository.

- GWWS server license

  The GWWS server requires an extra license from BEA to enable the functionality. Check to make sure it has been installed properly.

- GWWS server port configuration.

  Check the GWWS server port defined in the SALT configuration file. Avoid port conflicts with other applications.

- GWWS instance ID.

  Check the GWWS instance ID to make sure the two names are consistent in UBBCONFIG and SALT configuration files.

- UBBCONFIG file `MAXWSCLIENTS` definition.

  Make sure that `MAXWSCLIENTS` is defined in the `*MACHINE` section of UBBCONFIG file on the computer where GWWS server is deployed.

- `RESTART=Y` and `REPLYQ=Y parameters`.

  If the GWWS server is set to `RESTART=Y` in the UBBCONFIG file, `REPLYQ=Y` also must be defined.

- SALT configuration file.

  Make sure the SALT configuration file is accessible and set correctly for the GWWS server.

GWWS Rejects SOAP Request

In some cases, the GWWS server may reject SOAP requests. The most common causes are:

- WSDL document is outdated

  The WSDL document used by SOAP clients is out of date and some services may not be available.

- GWWS server environment variables are not set correctly

  When exporting a Tuxedo service with FML/VIEW buffers to a Web service, make sure the related GWWS environment variables are set with valid values. The GWWS server needs this information for the data mapping conversion.

- Violated Tuxedo Service Metadata Repository restrictions

  Check the SOAP client data and make sure Tuxedo Service Metadata Repository restrictions are not violated.

- Unavailable Tuxedo service

  Make sure the Tuxedo service you want exported as a Web service is available.

BEA SALT Message Tracing

The GWWS server supports Tuxedo `TMTRACE` functionality (used to dynamically trace messages). All trace points are logged in the ULOG file. Checking the ULOG file trace information helps to evaluate GWWS server SOAP message problems. GWWS server message tracing behavior is set using the `TMTRACE` environment variable, or by using the `tmadmin chtr` sub-command command.

The reserved trace category, `msg`, is used to trace BEA SALT messages. It can be used together with other general trace categories. For example, if trace category "`atmi+msg`" is specified, both BEA SALT and Tuxedo ATMI trace messages are logged.

Notes: Message tracing is recommended for diagnostic treatment only.

The following trigger specifications are not recommended for GWWS servers:

```
abort, system, sleep
```

In any of these trigger specifications are used, GWWS servers may be unexpectedly terminated.

For more `tmtrace` and trace specification information, see tmtrace(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

TMTRACE specification examples for BEA SALT message tracing are shown below:

- To trace SALT messages only
  ```
  export TMTRACE=msg:ulog:
  ```
- To trace both BEA SALT and Tuxedo ATMI messages
  ```
  export TMTRACE=atmi+msg:ulog:
  ```

Listing A-1 shows a ULOG file example containing BEA SALT tracing messages.

Listing A-1 Standard TMTRACE Messages

```
183632.BOX1!GWWS.4612.4540.0: TRACE:ms:A HTTP message is received, SCO
index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:A SOAP message is received, SCO
index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:Begin data transformation of
```

```
request message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:End of data transformation of
request message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.840.0: TRACE:ms:Delivering a message to Tuxedo,
service name =TOUPPER,  SCO index=1023

183632. BOX1!GWWS.4612.840.0: TRACE:ms:Got a message from Tuxedo, SCO
index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:Begin data transformation of reply
message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:End of data transformation of reply
message, buffer type = STRING, SCO index=1023

183632. BOX1!GWWS.4612.4540.0: TRACE:ms:Send a http message to net, SCO
index=1023
```

A more complex log is generated by `TMTRACE=msg:ulog`, used in WS-ReliableMessaging communication. All the application and infrastructure messages are sent to the ULOG file. Listing A-2 shows a ULOG file example containing WS-ReliableMessaging TMTRACE messages.

Listing A-2 WS-ReliableMessaging TMTRACE Messages

```
184706.BOX1!GWWS.3640.4772.0: TRACE:ms:A HTTP message is received, SCO
index=1023

184706.BOX1!GWWS.3640.4772.0: TRACE:ms:A HTTP Get request is received, SCO
index=1023

184706.BOX1!GWWS.3640.4772.0: TRACE:ms:Send a http message to net, SCO
index=1023
```

```
184710.BOX1!GWWS.3640.4772.0: TRACE:ms:A HTTP message is received, SCO
index=1022

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:A SOAP message is received, SCO
index=1022

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Create a new inbound sequence,
ID=uuid:4F1FEE40-72CB-118C-FFFFFFC0FFFFFFA8FFFFFFEB010000-1811

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Create a new outbound sequence,
ID=uuid:f7f76200-f612-11da-990d-9f37c3d14ba7

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Send CreateSequenceResponse message
for sequence uuid:4F1FEE40-72CB-118C-FFFFFFC0FFFFFFA8FFFFFFEB010000-1811

184710.BOX1!GWWS.3640.4772.0: TRACE:ms:Send a http message to net, SCO
index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:A HTTP message is received, SCO
index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:A SOAP message is received, SCO
index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:Begin data transformation of request
message, buffer type = STRING, SCO index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:End of data transformation of
request message, buffer type = STRING, SCO index=1022

184712.BOX1!GWWS.3640.3260.0: TRACE:ms:Received a request message in
sequence uuid:4F1FEE40-72CB-118C-FFFFFFC0FFFFFFA8FFFFFFEB010000-1811
```

WSDL Document Generated Incorrectly or Rejected by SOAP Client Toolkit

If the WSDL document does not generate correctly, or is rejected by the SOAP client toolkit, do the following:

- Try using the Doc/literal encoded style and SOAP 1.1 SALT WSDL document generation options. This is also the default behavior.
- Use `tmwsdlgen` to generate the WSDL document manually and compare with the one provided by GWWS online downloading. If the TMETADATA server is not started when the GWWS server booted, the GWWS server cannot obtain the correct service contract information. Therefore, the downloaded WSDL document does not contain the correct type definitions. You can use `wsadmin->creload` to dynamically reload the configuration file.