Retek® Integration Bus™ 10.3.4

Technical Architecture Guide

Corporate Headquarters:

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
USA

888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000

Fax:
+1 612 587 5100

European Headquarters:

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax:
+44 (0)20 7563 46 10

Printed in the United States of America.

Customer Support

Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Retek customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

Contact Method	Contact Information
E-mail	support@retek.com
Internet (ROCS)	rocs.retek.com Retek's secure client Web site to update and view issues
Phone	+1 612 587 5800

Toll free alternatives are also available in various regions of the world:

Australia	+1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus)
France	0800 90 91 66
United Kingdom	0800 917 2863
United States	+1 800 61 RETEK or 800 617 3835
Mail	Retek Customer Support Retek on the Mall 950 Nicollet Mall Minneapolis, MN 55403

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step by step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

Contents

Chapter 1 – Introduction

Welcome to the Retek 10.3 Integration Bus Technical Architecture Guide. This guide describes the technical architecture of the Retek Integration Bus (RIB). The goal is to illustrate the capabilities and issues an enterprise may encounter when integrating applications with the RIB. The intended audience for this guide includes system designers and project managers. It assumes that you are familiar with Enterprise Application Integration terms and concepts. If not, see the "Additional resources" section for more information.

Chapter 2 introduces the RIB message model. Important conceptual topics are presented such as the business event relationship to the message, the message 'family,' and message structures. Because the sequence of events that occur on a table reflect business processes, this chapter discusses the association of message structure and sequencing to systems and their availability on the RIB. Error handling, performance, and the synchronization of participating systems are topics touched on here. Finally, Chapter 2 presents the message lifecycle, or how messages flow through the system. Described are simple flows of messages that do not require additional transformation, filtering, or routing logic (known as a 'TAFR') to occur on the RIB, and those flows that depend upon a further TAFR operation prior to another application's subscription of the message.

The components of both SeeBeyond's e*Gate Integrator–the RIB itself–and Retek applications on the RIB are described in Chapter 3. Here you learn about SeeBeyond components like the registry, schema, event type definitions, e*Ways, intelligent queues, collaborations, and more. Because certain Retek applications have moved to the J2EE environment, this chapter also introduces the J2EE Enterprise Java Bean and Message Driven Bean components. Non-J2EE based Retek applications are characterized by the use of Oracle-based triggers and XML and Message Family manager packages for publishing messages through application adapters. Retek applications also share common message subscription processes for message and error handling. TAFR processing is presented too.

Learn about Retek Message Families in Chapter 4 where the event type and Message Family concept is discussed. Here you can see a list of Message Families for each application: Retek Merchandizing System (RMS), Retek Customer Order Management (RCOM), and Retek Distribution Management (RDM). If you are considering the interface of additional applications on the RIB, read Chapter 5. The successful coupling of third-party applications to the RIB (and, as a result, to Retek applications) hinges on understanding the importance of the single event-message relationship. These concerns are addressed here, along with descriptions of SeeBeyond proprietary e*Gate adapters that a client can select for applications to be deployed on the RIB.

Chapters 6 and 7 introduce the integration of Retek Extract, Transform, and Load (RETL) and batch file transmission on the RIB. RETL (extraction-transformation-load) is a framework you can deploy for high-volume data processing, especially in a multi-CPU execution environment. Both RETL and batch job integration involve the movement of files across the RIB. Currently, implementation of these processes involves further definition, and these chapters discuss the relevant issues.

Additional resources

Read the following Retek 10.3 and SeeBeyond documents for additional information.

Retek 10.3 integration documents

The following resources should be used for further understanding the Retek Integration Bus:

Retek 10.3 Integration Guide – Descriptions of Retek applications on the RIB and the functional areas in which they share data. The guide also contains all data descriptions, including the message catalog; XML document type definitions of messages; and mapping documents that specify a message's source application, table, column, and data type.

Retek 10.3 Integration Bus Primer – An introduction to basic Enterprise Application Integration (EAI) concepts and to the Retek Integration Bus (RIB).

Retek 10.3 Integration Bus Deployment Guide – Discussion of deployment considerations, design patterns, and strategies.

Retek 10.3 Integration Bus Installation Guide – Descriptions of:

- SeeBeyond e*Gate Integrator installation of its registry host and all participating host software, plus Graphical User Interface hosts for development and system monitoring.
- How to import the RIB schema into the e*Gate Integrator product.
- Configuring database connection points and JMS topics, updating CLASSPATH configuration values, and deleting unused adapters.
- Instructions for RIB components for applications using ISO or J2EE platforms.

Retek 10.3 Integration Bus Operations Guide–Provides a basic understanding of RIB components, how messages flow between them, and operational activities surrounding the components. Included are templates for using the RIB as an alternative to FTP batch jobs to transfer files from one system to another.

Retek 10.3 Extract, Transformation and Load (RETL) Programmer's Guide -- Provides information on using RETL for high-volume data extraction and loading.

SeeBeyond Technology Corporation documents

See the resources listed in this section to learn more about the RIB as it is deployed through the SeeBeyond e*Gate Integrator EAI platform:

SeeBeyond Business Integration Suite Deployment Guide – Information to use in analyzing, planning, and managing an EAI deployment.

SeeBeyond Business Integration Suite Primer – An introduction to all SeeBeyond e*Gate products, including e*Ways for popular applications like:

- PeopleSoft
- SAP
- Oracle Financials

Chapter 2 – The RIB messaging model

This chapter presents the RIB's messaging model. It describes how RIB messages are structured and the rationale behind this structure. It also describes the types of messages used.

Not presented in this chapter are the specifics of each message. The Retek 10.3 Integration Guide details information about message contents and transformations.

Message characterization

Enterprise Application Integration systems produce messages characterized by three dimensions: the contents of the message, when the message is produced, and the structure of the message.

Note: The term "message characterization" is used as opposed to "message type" to avoid confusion with other EAI terms.

Structure: The message may have a simple structure and correspond to a small business sub-entity or it may contain a hierarchical structure containing all sub-entities that comprise it. ("Flat" versus "hierarchical".)

Message contents: The message contains all information about a business entity or it captures only something that has changed about that entity ("snapshot" versus "delta").

When the message is produced: The message may be produced as part of the business transaction affecting the entity or it may be produced within a separate transaction that occurs a short period of time later. ("Synchronous" versus "asynchronous" production.)

Using these criteria, one is able to characterize a specific message as a "flat synchronous snapshot" or a "hierarchical asynchronous delta" or a "hierarchical synchronous snapshot" or some other combination. Additional information accompanies the business entity information. This includes XML tags used to rout the message, information about the originating system or environment, or information about the business event the message captures.

The RIB publishes three different message characterizations:

- Hierarchical Synchronous Snapshots – These messages contain newly created composite business entities, such as purchase orders.
- Flat Synchronous Snapshots – These messages contain a change made to a business entity absolute value, such as the price of an item, on a "master" system. They may also contain newly created simple business entities, such as a location.
- Flat Synchronous Deltas – these messages encapsulate a business event captured on a non-master system that affects information on a remote "master" system. An example of this would be for a clerk to reserve inventory for a local store system from a remote warehouse system. The remote warehouse system is the master of its inventory data.

RIB Message Families and Message Types

Besides the characterizations of a message, each RIB message belongs to a specific *Message Family*. Each Message Family contains information specific to a related set of operations on a business entity or related business entities. The publisher is responsible for publishing messages in response to actions performed on these entities in the same sequence as they occur.

Descriptions of each Message Family are found later in this document. Although a generalized format exists, each Message Family varies in the specifics of the information it contains – the business entities and events the message captures. Furthermore, each Message Family contains a set of sub-formats specific to the business event triggering message publication. The term *message type* embodies this specific sub-format. For example: a Purchase Order Message Family can contain Message Types such as “Create PO Header”, ”Create PO Detail”, “Update PO Header”, or “Delete PO Detail”.

Messages are published and subscribed to on a Message Family basis. A single application is responsible for publishing all messages within a Message Family. However, multiple instances of an application may publish messages within the same Message Family. In other words, only the RMS application publishes messages in the “Available To Promise” (ATP) Message Family and only the RDM application publishes messages in the “Advanced Ship Notice Outbound” (ASN Outbound) Message Family. However, multiple distribution center installations of RDM may each publish their own ASN Outbound messages.

Model drivers and concerns

An architect chooses the type, structure, and other characteristics of the messages within an EAI system based upon many factors. One major factor is how the message contents encapsulate a business event. Different characterizations are available within a single EAI system. The RIB is no exception. The RIB contains many messages characterized as “Hierarchical snapshots” and “synchronously” produced. On the other hand, there are also “flat synchronous delta” RIB messages associated with update operations. The factors determining which characterization to use include:

- **Publisher/subscriber/bus availability:** One major goal in the design of the RIB is to insure that no tight coupling exists between Retek’s applications and the RIB’s availability. That is, if the RIB is unavailable, the publishing and subscribing applications can still function. This means that there may be a delay before the transmission of a message occurs over the RIB network. It also means that database updates needed for message publishing must occur outside of the same transaction containing the business event.
- **Retek application locking on sub-business entities:** Many of Retek’s applications allow for simultaneous updates to sub-business entities. An example of such an entity is a line item found within a Purchase Order. The Retek Merchandising System allows multiple concurrent changes to multiple items, header, or summary information for a single PO. Many times the PO is used for replenishment purposes and multiple people are constantly updating the PO. Situations such as these tend to produce “flat” messages containing only the changes to the line items. Producing a “hierarchical” message would risk locking the PO for an unacceptable amount of time.
- **Concurrency of message contents production and business event:** A desire for a loose coupling between the RIB and the business application suite drives some EAI architectures. In many cases, message information is staged before publication. A delay exists between when the business event occurs and when the message corresponding to this event is created and published. This delay presents a window of opportunity for multiple similar business events to occur on the same entity before publication of any of the messages. For example, multiple users may make changes to the same Purchase Order header within a short time period.

  There are two strategies for staging business event information: record only enough information to denote that the event occurred (for example: an update occurred on PO line item #123) or record all information about that event (for example: an update occurred on PO line item #123 and the new quantity is 4, the new location ID is 8,). If only some of the information is staged, the message published may not correspond to the triggering business event. In this case, the publisher assumes that the subscriber is interested only in the resultant business object and has little or no interest in datum such as the number of times a change has occurred.
- **Transactional considerations:** Some business events require multiple database transactions to complete. One example of this is the creation of a new vendor**.** In this case, all of the surrounding foundation data must be present before the vendor specifics. This foundation data includes information such a valid country code identifying the vendor’s country of origin, one or more valid currencies, and other specific terms, conditions or other policy identifiers used to conduct transactions with the vendor.

- **Sequencing and error handling:** Many business processes are stateful. That is, only certain actions can occur at certain times. A subscriber must process messages concerning a specific business entity in the same order they were published. This has implications regarding error handling: once an error occurs on one message, subsequent messages referring to the same business object should be held and not processed until the error has been resolved. However, other messages concerning other business entities should continue to be processed.
- **Deployment and software lifecycle:** The applications producing and subscribing to messages need separate deployment between themselves and the RIB. In effect, each Retek application can be "plugged" into the RIB based on the needs of the retailer. If the retailer decides to not use the RIB, then no noticeable performance degradation occurs. In other words, the RIB is not required for any Retek application to function in a stand-alone manner.
- **Performance:** Updates to some business sub-entities happen frequently on a single business entity. Take the example of a retailer creating a single replenishment PO per supplier. Users may update the same PO many times during the day. When one analyzes the volume of updates and the cost of creating a full PO message, it may be a significant performance bottleneck to publish the full PO snapshot for each update.

  Another performance consideration is the granularity of a message and the requisite overhead to process the contents of a message. This includes the following factors:

  - Per-message overhead – the amount of processing needed to simply retrieve a message from the associated message server and to perform a two-phase commit operation.
  - Retrieval of referenced data – the external data needed to process a message that is referenced, but not contained, within the message.
  - Aggregation of contents – the number of logical units and their contents contained in a message. Aggregation is a performance enhancement technique that allows more data to be processed in a single physical unit of work by spreading overhead among many logical units of work.
- **Scalability:** Associated with performance is how well the system can scale. Scaling concerns come to the forefront when a single thread of processing cannot perform well enough to process a required amount of data. Ideally, a scalable application will perform in a linear manner according to the available resources – doubling the number of processing instances and resources should double the throughput of the application. The main concern for scalability is inherent in the resource contention between threads. These concerns can only be addressed by the message definition and the associated database locks held while processing a message. In certain circumstances, a message may be processed by an application in multiple database transactions to insure scalability.
- **Data synchronization risks:** Many messages seek to replicate data across multiple systems. Sometimes, the data on two systems may differ due to a variety of possible situations. When one uses a "delta" type of message, there is a risk that the subscriber cannot process these messages due to the data differences.

Message life cycle

The Retek Information Bus (RIB) uses the "Pub/Sub" message model for all of the messages produced and consumed within the EAI system. The publishing application is responsible for creating the initial message contents. The RIB publishing adapter will publish it to the RIB and make it available to any subscribers. The RIB knows what subscribers are to receive the message due to the RIB's configuration -- this configuration associates a set of subscribers to each publisher / Message Family combination.

Database tables associated with the publishing application typically stage message information. On the SeeBeyond platform, one or more RIB Publishing Adapter collaborations poll the application via a stored procedure call. A collaboration is a single thread of control within the adapter. On the J2EE platform, the application calls a Retek developed Enterprise Java Bean (EJB) with the payload information to be published.

The message resides on a Java Message Service (JMS) topic[1] immediately after publication. The JMS topic provides stable storage for the message in case a system crash occurs before all message subscribers receive and process it.

One system requirement is that a message must be delivered to and processed successfully exactly once by each subscriber. Furthermore, all work performed by the subscriber and the RIB must be atomically committed or rolled back, even if the JMS server is on a remote host. The standard way to perform this is by using an XA[2] compliant interface and two-phase commit protocol.

After initial publication, a message may undergo a series of transformation, filtering, or routing operations. A RIB component that implements these operations is known as a Transformation and Address Filter/Router (TAFR) component. A *transformation operation* changes the message data or contents. A *filter operation* examines the message contents and makes a determination as to whether the message is appropriate to the subscriber. For example: those subscribers that do not process all Message Types found in a Message Family require filter operations to weed out the unsupported types. A *router operation* examines the message contents and forwards the message to a subset of its subscribers. A filter operation can be considered a special case of a routing operation. Although logically separate operations, for performance reasons TAFR components usually combine as many as is appropriate.

TAFR operations are specific to the set of subscribers to a specific Message Family. Multiple TAFRs may process a single message for a specific subscriber and different specific TAFRs may be present for different subscribers. Different sets of TAFRs are necessary for different Message Families.

If all subscribers to a message can process all messages within a Message Family without any TAFR operations, then no TAFR components are needed, as seen in Figure 2.1.

[1] A "JMS topic" is a queue of messages that can be shared between multiple subscribers and each subscriber can independently access every message on the topic. A "JMS queue" is a queue of messages which, if shared between multiple subscribers, allows for only one subscriber to see any specific message.
[2] XA is a standard specification that details the interface between multiple "Resource Managers" and a "Transaction Manager". It insures that distributed transactions are performed correctly within a heterogeneous environment.

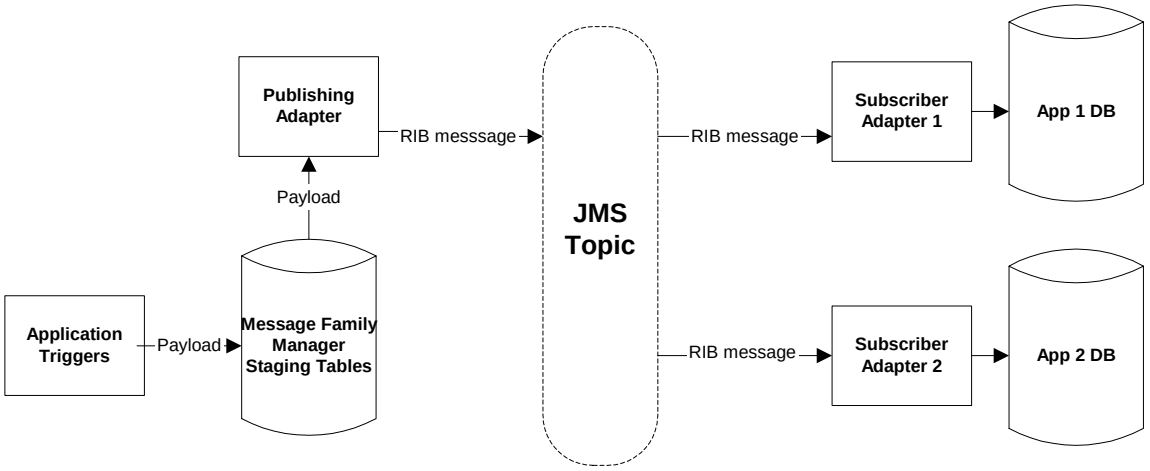


Figure 2.1 Simple Message Flow

Multiple TAFRs may be needed depending on the types of subscribers. This is seen in Figure 2.2, where one TAFR routes the information among different remote sites and then another TAFR transforms the data further for an additional subscriber.

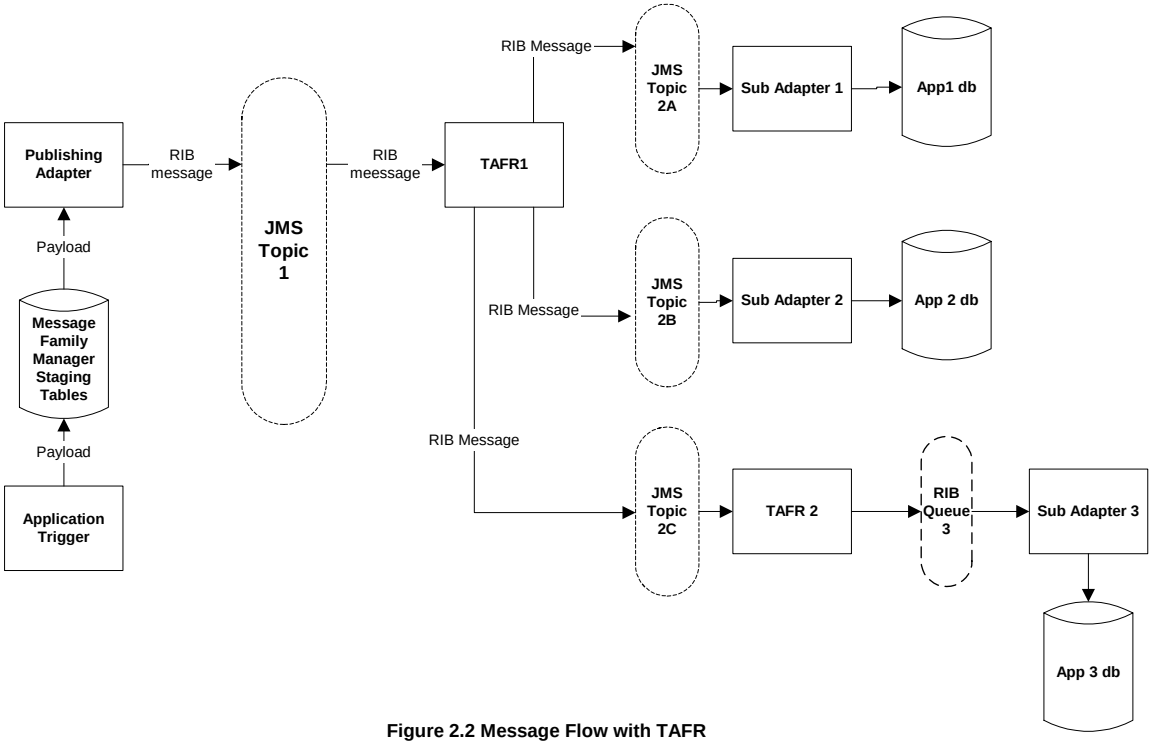


Figure 2.2 Message Flow with TAFR

Another type of RIB component that may process a message is a *bridge* component. These SeeBeyond e*Ways, BOBs, queues, or connection points allow messages to traverse different administrative domains. The type of bridge component used is site specific. A deployment of bridge components is dependent on how the network bandwidth and topology, the administrative specifics of the publisher and subscriber applications, and the availability of specific RIB resources. Bridges are very useful when remote sites that belong to different organizations and operations staff need to exchange messages and a central controlling authority is non-existent. Figure 2.3 is a modification of Figure 2.2, where one of the remote systems uses a bridge.

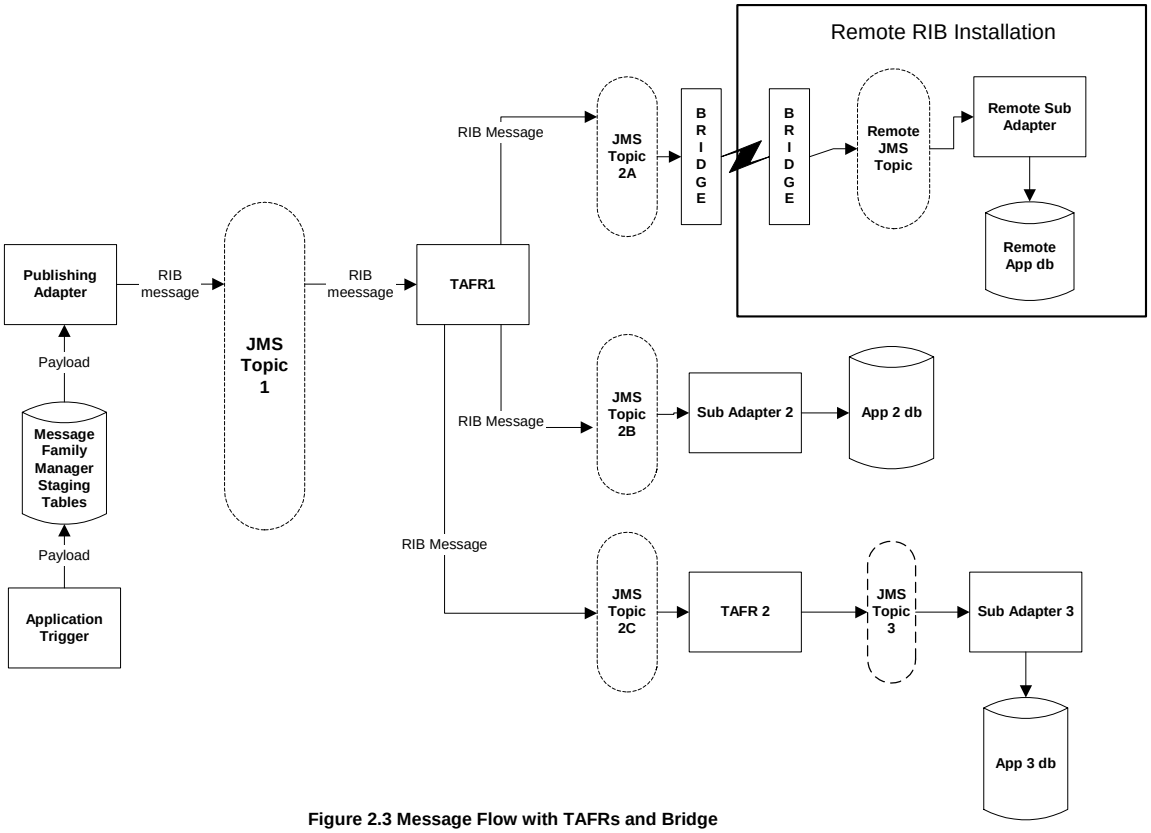


Figure 2.3 Message Flow with TAFRs and Bridge

Within RIB components, message processing continues until a subscribing adapter successfully processes the message. These components will perform application specific database updates for the specific message encountered.

When a message is processed, the adapter checks to see if the Error Hospital contains any messages associated with the same entity as the current message. If so, then the adapter places the current message in the hospital as well. This is to insure messages are always processed in the proper sequence. If proper sequencing is *not* maintained, then the subscribing application will contain invalid data.

If an error occurs during message processing, the subscribing adapter notes this internally and rolls back all database work associated with the message. When the message is re-processed (since it has yet to be processed successfully), the adapter will now recognize this message is problematic (sick) and checks it into the hospital for a “cure”.

After a message is checked into the Error Hospital, a second collaboration extracts the message from the hospital and re-publishes it to the integration bus. The message remains in the hospital during all re-tries until the subscribing adapter successfully processes it.

RIB message structure

RIB Messages are XML formatted. Multiple business events may be aggregated or bundled into a single message. The outer tag, <RibMessages> may contain multiple <ribMessage> tags, each of which represents a separate business event. The <RibMessages> tag may also contain a single <publishetname> tag. When the source of data is a file, certain collaborations use this tag to determine the correct event type (JMS Topic) to publish the message to. It is only valid when a file must be loaded as a single message using the RIB's generic file loading collaboration rule. Each <ribMessage> tag is a two-tiered structure consisting of a set of "envelope" tags and a single "payload". The envelope tags contain routing, message type, and other non-business entity information. The payload is specific to the message type and contains the business entity information.

As of the RIB 10.3 release, the message envelope contains the following tags:

`<RibMessages>`
root message tag. This tag contains one or more <ribMessage> tags.

`<ribMessage>`
tag delimiting information regarding a single event that has occurred on a business object. This tag contains all of the elements below:

`<family>`
Message Family message belongs to

`<type>`
message type message belongs to

`<id>`
Optional ID string that identifies the message. Composite primary keys will require multiple IDs. For example, a line item within a Purchase Order would contain the PO number and line item number as part of the ID. For example:

```
<id>PONumber=12345</id>
<id>ItemID=321</id>
```

Some ID's are simple and the value of the ID is specific to the Message Family. In this case, a single ID tag may be present and consist of merely a single identifier, such as `<id>FT_ITEM_12</id>.`

<routingInfo>
Optional tag that contains elements used to route or filter messages for specific subscribers. Multiple <routingInfo> tags may be present. Within the <routingInfo> element, the following sub-elements must exist:

<name>
name of routing field. A message may have multiple routing fields.

<value>
value of the routing field.

<detail>
optional tag containing additional qualification of the name/value. There may be up to two <detail> tags found within each <routingInfo> tag. Sub-elements of <detail> are:

<dtl_name>
name of the detail field.

<dtl_value>
value of the detail field.

The values of the <name>, <value>, <dtl_name>, and <dtl_value> are specific to the message family.

<publishTime>
Date/timestamp the message was published. Must be in the form, yyyy-MM-dd HH:mm:ss.SSS z

where:

yyyy is the year
MM is the numeric month (1 – 12)
dd is the day of the month
HH is the hour of the day (0 – 23)
mm is the minutes of the hour (0 – 59)
ss is the seconds of the minute (0-59)
SSS is the milliseconds of the second (000 – 999)
z is the three character time zone specification

<hospitalID>
This is an optional element. It constains the ID of the Message within the Error Hospital. Must be set when the message has been resubmitted or retried.

<failure>
Optional tag that contains elements used to identify a specific processing error. Multiple <failure> tags may exist. Every time the message is checked into the Error Hospital, a <failure> tag is created. This tag contains the following sub-elements:

<time>
Date/timestamp of failure.

<location>
Location or name of the Error Hospital.

<description>
Textual description of the error.

`<messageData>`
The message type specific "payload" containing data describing the message triggering event. The payload is XML, but the XML varies within each message type. The DTDs describing this data are stored in a table within the rib_message database table.

`<ribmessageID>`
This field uniquely identifies the message based on the publishing adapter. It may be used to track or correlate problems associated with a specific message.

`<customData>`
Optional field reserved for client specific additions to RIB message payloads.

`<customFlag>`
Reserved for future use. Must be set to 'F'.

`<hospitalRef>`
This is an optional field. The reference to a hospital record used by custom post-processing in conjunction with the hospital controller. It allows a successful completion of one message to cause a change in a status of a message in the error hospital, so that potentially it can be retried. The hospitalRef contains 4 components:

- MessageNum (Unique hospital ID)
- Message Family
- Old Reason Code
- New Reason Code

Sample RIB Message

The sample RIB Message below contains a <RibMessages> tag containing two <ribMessage> nodes. The <messageData> tag contains data for a warehouse create and modification messages. Routing information has been added for this example; this message does not normally contain routing information.

Also in the example, one difference between each <ribMessage> node is the format of the <messageData> string. This tag contains XML tags itself. Tag delimiters and quotation marks within an XML tag must be changed or surrounded by a CDATA declaration in order for it to be well-formed. The first node uses a CDATA declaration. Using the CDATA declaration is more efficient than replacing the XML reserved characters. However, it requires that the string itself never contains the ending delimiter for a CDATA string, ">>]". The second changes the reserved XML characters using the "<", ">", and """ for "<",">" and """ (double quotation), respectively[3]. Either format may be used in a <messageData> element.

Whitespace between different XML elements is optional. However, whitespace should not be found immediately following the <messageData> tag and the message payload itself.

[3] There are two additional reserved characters in XML, "&" (ampersand) and "'" (apostrophe or single quotation mark). Their replacement strings are "&" and "'".

```
<?xml version="1.0" encoding="UTF-8" ?>
<RibMessages>
  <ribMessage>
    <family>WH</family>
    <type>WHCre</type>
    <id>22</id>
    <ribmessageID>10.3|ewWHFromRMS|colWHFromRMS|2003.05.26
13:43:29.123|78</ribmessageID>
    <routingInfo>
      <name>to_phys_loc</name>
      <value>9901</value>
      <detail>
        <dtl_name>to_phys_loc_type</dtl_name>
        <dtl_value>S</dtl_value>
      </detail>
    </routingInfo>
    <publishTime>2003-05-26 18:06:29.809 CDT</publishTime>
    <messageData><![CDATA[<!DOCTYPE WHDesc SYSTEM
"http://www.retek.com/dtd/rib/WHDesc.dtd">
<WHDesc>
   <wh>22</wh>
   <wh_name>WH1</wh_name>
   <wh_add1>19 Pruneridge Ave</wh_add1>
   <wh_add2/>
   <wh_city>Cupertino</wh_city>
   <county/>
   <state>CA</state>
   <country_id>USA</country_id>
   <wh_pcode>95014</wh_pcode>
   <email/>
   <stockholding_ind>Y</stockholding_ind>
   <channel_id/>
   <currency_code>USD</currency_code>
   <duns_number/>
   <duns_loc/>
   <physical_wh>1</physical_wh>
   <break_pack_ind>Y</break_pack_ind>
   <redist_wh_ind>N</redist_wh_ind>
```

```
   <delivery_policy>NEXT</delivery_policy>
</WHDesc>]]></messageData>
    <customFlag>F</customFlag>
  </ribMessage>
  <ribMessage>
    <family>WH</family>
    <type>WHMod</type>
    <id>22</id>
    <ribmessageID>10.3|ewWHFromRMS|colWHFromRMS|2003.05.26
13:43:29.123|79</ribmessageID>
    <routingInfo>
      <name>to_phys_loc</name>
      <value>22</value>
      <detail>
        <dtl_name>to_phys_loc_type</dtl_name>
        <dtl_value>S</dtl_value>
      </detail>
    </routingInfo>
    <publishTime>2003-05-26 18:06:29.834 CDT</publishTime>
    <messageData>&lt;!DOCTYPE WHDesc SYSTEM
&quot;http://www.retek.com/dtd/rib/WHDesc.dtd&quot;&gt;
&lt;WHDesc&gt;
   &lt;wh&gt;22&lt;/wh&gt;
   &lt;wh_name&gt;WH1&lt;/wh_name&gt;
   &lt;wh_add1&gt;20 Pruneridge Ave&lt;/wh_add1&gt;
   &lt;wh_add2/&gt;
   &lt;wh_city&gt;Cupertino&lt;/wh_city&gt;
   &lt;county/&gt;
   &lt;state&gt;CA&lt;/state&gt;
   &lt;country_id&gt;USA&lt;/country_id&gt;
   &lt;wh_pcode&gt;95014&lt;/wh_pcode&gt;
   &lt;email/&gt;
   &lt;stockholding_ind&gt;Y&lt;/stockholding_ind&gt;
   &lt;channel_id/&gt;
   &lt;currency_code&gt;USD&lt;/currency_code&gt;
   &lt;duns_number/&gt;
   &lt;duns_loc/&gt;
   &lt;physical_wh&gt;1&lt;/physical_wh&gt;
```

```
   &lt;break_pack_ind&gt;Y&lt;/break_pack_ind&gt;
   &lt;redist_wh_ind&gt;N&lt;/redist_wh_ind&gt;
   &lt;delivery_policy&gt;NEXT&lt;/delivery_policy&gt;
&lt;/WHDesc&gt;</messageData>
    <customFlag>F</customFlag>
  </ribMessage>
</RibMessages>
```

Chapter 3 – Messaging system component overview

This chapter details the major components of the RIB that create, process, or consume messages.

The 10.3 release of the RIB has a diverse set of application interfaces. For some Foundation Data interfaces, the 10.3 RIB release uses Character Large Object Binaries to communicate with the Oracle Stored Procedures. For high-volume messages, the interfaces to Oracle Stored Procedures use a RIB specific set of Oracle Objects. Another variant in use has the RIB infrastructure implemented with the Java 2 Platform, Enterprise Edition (J2EE) environment to work with Retek Applications deployed within the J2EE environment.

SeeBeyond components

The RIB deployed on the SeeBeyond e*Gate Integrator platform uses an application provided Oracle Stored Procedure interface to process message payload or to create payloads for new messages. In this environment, the RIB components execute within the context of SeeBeyond's e*Gate Integrator framework. This section presents a brief overview of the associated components.

Registry

The e*Gate Registry is a SeeBeyond proprietary database containing all entities used within a running e*Gate system. There is at least one registry available to SeeBeyond components at all times. A system designer designates one registry as the "master". Other, "secondary" registries replicate the master for increased performance and system availability.

Schemas

A schema is a logical grouping of SeeBeyond EAI components. Each registry contains at least one or more schemas. Typically, schemas are designed for the end-to-end processing of a set of related messages. The design of a Schema within a deployed RIB system is dependent on many site-specific factors. Specific design or configuration options are discussed in the RIB Deployment Guide.

Control brokers and participating hosts

The control broker is responsible for maintaining the operational control and status of its attached components. Another goal of a control broker is to minimize the number of network connections to the registry and to provide a central point of control for a set of components. Each control broker connects to one registry but can also fail over to other registries if needed. The control broker and all of the attached components must belong to a single e*Gate schema.

There is one control broker per “participating host” per SeeBeyond e*Gate schema. A participating host is a logical construct used. The control broker’s TCP/IP address and the participating host’s name are associated with each other within the registry.

Control Brokers and participating hosts are transparent or not involved in the processing of RIB messages.

Events and event type definitions

SeeBeyond “events” include both messages passing to and from JMS, and stored procedure calls to external application APIs. An event’s type determines its logical name, but the rules for parsing are determined by an event type definition (ETD). Hence, the ETD has a strong coupling with the message structure. Different event types may share the same ETD to allow message with identical structure to flow to different recipients. The RIB uses a single ETD for all messages while they are inside the RIB.

Collaborations

Collaborations define message processing logic on a per Message Family/message source/ component combination. This logic is “triggered” or executed when the adapter pulls a message with the correct event type from the specified source. The RIB uses Java to define the message processing logic. All collaborations require one or more triggering conditions in order to execute. This condition may be any of the following:

- A file appearing in some directory
- A certain time period has elapsed
- A message appearing on a queue
- Some application – specific condition

A collaboration works on a collection of input and output events, which may be messages going to or from queues, or passing to or from an application's RIB APIs.

In general, the logic within a collaboration may perform any number of operations. It may update a database, simply collect statistical data, write information to a file, or some other operation. It may produce zero or hundreds of output events, depending on the application.

e*Ways and BOBs

There are two basic types of e*Gate components used to create, process, and/or consume messages on the RIB: e*Ways and Business Object Brokers (BOBs). These are specific implementations of the generalized concept known as an Integration Bus "Adapter". BOBs and e*Ways contain one or more "Collaborations" that are triggered from some event. A collaboration works on a collection of input and output events, which may be messages going to or from queues, or passing to or from an application's RIB APIs.

Note: See the Retek Integration Bus Primer if you are unfamiliar with the concept of an Integration Bus Adapter.

e*Ways and BOBs are multi-threaded and can process multiple messages simultaneously, but are single-threaded for a particular event type.

Traditionally, the difference between the two component types is that e*Ways may contain an "application specific" source or sink for messages, while BOBs connect internal bus components. The RIB, however, only uses a specific type of e*Way, the Java "Multi-mode" e*Way, which can function as both an external source or sink and an internal connector. The Multi-mode e*Way is a grouping of logical collaborations into a single physical process or program.

Intelligent Queues and JMS Intelligent Queues

Intelligent Queues (IQ) hold published messages and maintain a record of what subscribers have received the messages. Many types of Intelligent Queues either wrapper the message storage mechanism or bridge to another queuing system. The SeeBeyond e*Gate system installed with the RIB includes a Java Messaging Service (JMS) IQ. JMS Intelligent Queues are queues that may be accessed using the Java Message Service API.

IQ Managers and JMS IQ Managers

One primary purpose of an Intelligent Queue Manager is to control a set of Intelligent Queues of the same type. There are multiple types of Queue Managers, each controlling a different type of IQ. Each type of IQ differs on how messages are queued and saved to stable storage while in the queue.

The JMS Intelligent Queue Manager serves two roles. The first is the same as any other IQ manager: to control a set of Intelligent Queues for any SeeBeyond e*Way. The second (which the RIB uses) is to act as a Java Message Service (JMS) provider, accessible through JMS Connection Points. The RIB uses the IQ Manager this way because it requires the use of the XA two-phase commit protocol to guarantee "exactly once" successful message processing. This protocol is available with a JMS implementation. However, a JMS Intelligent Queue is not used because the existing IQ Manager service interface does not support the XA protocol. Instead, RIB e*Ways use SeeBeyond JMS Connection Points. Connection Points connect to a JMS IQ Manager such that the XA protocol is supported. For more information regarding JMS connection points and Intelligent Queues, see the *SeeBeyond JMS Intelligent Queue User's Guide.*

The RIB is designed to only retrieve and publish messages to a JMS compliant server. The preferred JMS implementation is the SeeBeyond's standard JMS implementation. As of the 10.3 release, Retek has not certified other JMS implementations or interfaces.

e*Way Connection Points

An "e*Way Connection" or "Connection Point" defines a session between the e*Way and an external system. The following types of connections are available:

- Java Message Service – a connection to a JMS Server or JMS Service.
- A relational database, such as Oracle
- A TCP/IP connection to a remote application using the HTTP or HTTPS protocol.
- E-mail (uses standard SMTP for outbound and POP3 interfaces for inbound messages)

If a database connection point used within a collaboration defines the login, password, and server address for database operations. It also may define the frequency "triggering events" are fired off, allowing the collaboration to define a polling operation.

A connection point made to a JMS implementation can be used to publish or subscribe to external applications. JMS connection points can also be used to bridge between e*Gate schemas.

J2EE components

The Java 2 Platform, Enterprise Edition (J2EE) is a multi-tiered client/server architecture that allows an application to be deployed as a set of reusable components within a distributed processing environment. Client tier components run on a client machine and business tier components run on the J2EE server and database components run on a database server.

Retek applications that are deployed on the J2EE platform and integrating using the RIB will require the Retek Binding, Retek Message Driven Bean (MDB), and Enterprise Java Bean (EJB) components. Retek applications deployed using Oracle Forms do not have J2EE dependencies, except for a Java Message Service provider.

Please see Chapter 8 for more information on the RIB J2EE architecture.

Java Message Service Usage

The J2EE Java Message Service (JMS) specification provides a standard API used by RIB components for publishing and subscribing messages. This section details what parts of the specification are used.

In the e*Way environment, sending messages to and retrieving messages from the JMS is wrappered by a set of SeeBeyond proprietary classes. However, it could be possible to circumvent these classes, at the cost of additional program complexity. This means that the actual implementation is still JMS compliant.

For the RIB, all messages are published to a JMS Topic. The specific topic used is dependent on the Message Family the message belongs to and the current stage in the processing of the message. For example, the name of the topic used to hold messages pulled from RMS with vendor information is “etVendorFromRMS”. TAFR adapters may both subscribe to and publish messages in the same Message Family. In these cases, the re-published messages are put onto another topic.

The list of JMS topics used by RIB components is detailed in the RIB Integration Guide.

JMS Selectors

Another aspect of the JMS usage is the attachment of message properties to published messages and the use of selectors by message subscribers. Message selectors are used by the RIB to distinguish the desired subscribers for a message. The standard set of message properties are:

threadValue– the logical thread value associated with the multi-threading of a message stream. All messages for a specific business object will always contain the same threadValue property. This, combined with the standard FIFO message ordering on the topic, is integral to message sequencing. Messages with different threadValue properties are not guaranteed to be processed in the same relative order as publishing.

retryLocation -- This identifies a specific subscriber that is to retry this message. This property is only set when a message is currently in the Error Hospital and is scheduled for another attempt to be processed. It insures that messages being retried are only picked up by the original subscriber for those topics having multiple subscribers.

groupKey – This property identifies a group of subscribers for processing the message. The value of this property is an identification of a level within a hierarchy that is to receive this message. It is present for compatibility with the Retek Integrated Store Operations (ISO) platform.

Messages published without any selectors present will not be picked up by the standard RIB adapters in the SeeBeyond platform. By default, the RIB creates a selector that subscribes to messages:

- with a *threadValue* of '1' and
- a *retryLocation* of '<ewayName>.<collaborationName>' or null (not present).

Message Selector Check

Because these message selectors help guarantee single message processing and thread routing, we need to make sure that message selectors are properly set on each durable subscriber. Upon starting each e-way that subscribes to a message on a topic (Subscribers and Tafrs), the e-way checks its own selector and if it isn't set correctly it will check to see if any messages are awaiting consumption by that subscriber, and if no messages are waiting the durable subscriber is deleted and recreated with the proper message selector. However, if messages are waiting to be consumed the eway will explain this state and shut itself down before consuming any messages telling the user (in the RIBLogs) to extract the messages and fix the selector, then re-add the messages to ensure that the eway can make sure the messages are to be processed or if it should be filtered out.

This check and termination of the eway can be bypassed by changing a setting in the rib.properties file:

default.MessageSelectorCheck=true (change to false if you want to skip this validation).

Subscriber Check

In JMS, a publisher can publish a message without a subscriber available to listen to the message. In this case, the JMS vendor will silently throw away the message. This goes against the RIB requirement that no messages are lost. The JMS API specification does not contain anything regarding querying current subscribers and is considered an administration activity. Our solution is to utilize the administrative commands available in the JMS implementation to query subscribers at initialization for publishers and runtime for TAFRs.

When a publishing e*Way is started, the corresponding publisher helper is initialized, this happens once during the lifecycle of the Java Virtual Machine. During this initialization, a JMS utility command is run (e.g. stcmsctrlutil for SeeBeyond). The output for the command is then parsed looking for the number of subscribers. If this number is non-existent or 0, the publisher then proceeds to shut down so that no messages can be published. A corresponding exception is thrown to halt execution of collaboration logic. The logic is the same for a TAFR, but this occurs at run-time (during a message publish), because the topic to verify subscribers for is not known until a message is published. To aide in performance, TAFR's maintain a cache of topics already verified and only runs the JMS utility command if it hasn't already.

This check and termination of the eway can be bypassed by changing a setting in the rib.properties file:

default.SubscriberCheck=true (change to false if you want to skip this validation).

Enterprise Java Beans (EJBs)

Enterprise JavaBeans are a means to deploy application components without the developer necessarily worrying about low-level implementation details such as threading, transaction control, and load balancing.

EJB’s are deployed within a J2EE Server container. It is the container’s responsibility to instantiate an EJB, provide a thread of execution and perform load balancing. Depending on how the EJB is deployed and used, it may also be the container’s responsibility to provide the transactional context of calls made to the EJB.

There are two characteristics of EJBs: Session versus Entity and Stateless versus Stateful. All RIB EJBs are Stateless Session Beans. This means that these Beans are not associated with a specific database entity, but maintain a session with the client. It also means that no state is preserved between bean activations.

Message Driven Beans (MDBs)

Message Driven Beans are used to process messages from one or more Java Message Server Providers. The J2EE server is responsible for reading the message from the JMS provider and delivering it to the MDB onMessage() procedure. The Application developer creates the onMessage() method of the bean to implement all application specific logic.

For the J2EE deployment of the RIB, all of its MDB's begin by implementing the same code. This is because a) the MDB's deployment descriptors describe the RIB interface enough for Message Family specific processing and b) the Retek Binding Code enables the means for Retek application specific processing. The Retek Binding code is discussed later.

Deployment Descriptors

The deployment of EJBs and MDBs are through XML files known as Deployment Descriptors. Deployment Descriptors describe the attributes of a J2EE component in regards to what the component is, the number of instances allowed and the transactionality of a request made to the component.

Each application server has unique variances from other application servers in the available and required XML tags found in its deployment descriptors. Hence, Jboss deployed EJBs use a slightly different deployment descriptor than WebSphere specific deployment descriptors. Fortunately, tools exist to easily create application server specific deployment descriptors.

Deployment descriptors also specify the selector a MDB is using. Standard RIB messages will have a JMS message property, *threadValue,* set to a value defining a logical processing thread. By default, *threadValue* has a value of '1'. For messages being retried from the Error Hospital, an additional property, *retryLocation,* is set to make sure only the original subscriber will receive the message. Hence, most MDBs will have a selector of the form:

```
threadValue='1' and (retryLocation is null or retryLocation =
'<mdbID>' )
```

Where <mdbID> is the so-called locationID found in the 'location' column of the error message.

One very important aspect of the deployment descriptors for RIB MDB's is the control of the number of MDB instances and the number of messages retrieved from the JMS server at a single time. The J2EE specification allows multiple MDB instances to retrieve multiple messages at a single time from a specific JMS topic. The reason for this is to improve performance. However, if one simply increases the number of MDB's reading from a topic or the number of messages retrieved from the JMS, windows of opportunity arise for messages to be processed out of order. Hence, the RIB requires that each and every MDB deployed use at most one instantiation. Multi-threading the message processing must be done using separate deployment descriptors which specify different JMS "Selectors" for each deployed MDB for a single Message Family. This will insure that all messages for a single business object will always be processed in the correct order.

Transaction Managers

All RIB publishers and subscribers use an XA compliant two-phase commit operation to insure that

- A message is published if and only if the associated database transaction is successfully committed.
- A message is removed from the JMS server if and only if the associated processing is successful.

An integral part of this is a J2EE component known as a Transaction Manager (TM). Transaction Managers have been around for at least as long as the XA specification and are an integral part of three-tier client/server computing. The purpose of a TM is to start, end, and control a transaction involving multiple resources such as databases and JMS Servers.

EJB's and MDB's for the RIB are configured to use container managed transactions. This means that the J2EE container which hosts the bean controls the transaction either by implementing a TM or by using a TM implemented elsewhere in the application server.

It should be noted that all RIB Bean components must require the use of a global transactions. For MDB's, the transactionality of requests should be set that a Transaction is required and for RIB EJB's, the transactionality of method calls should be set to Mandatory. Otherwise, a window of opportunity exists whereby either a message is published twice to the JMS topic or the message is lost.

Integrated Store Operations (ISO) components

The ISO platform is a low-cost Retek application server available for use in store systems. Although this platform is not J2EE compliant, it is extremely similar to J2EE. Although some differences in terminology exist, such as the use of the term "ISO component" versus "EJB" or "MDB", the same basic paradigm is used to describe the architecture. A critical component is the usage of an XA compliant two phase commit. This insures that messages are removed if and only if a successful processing has occurred.

For those applications using the ISO platform, the Java Open Transaction Manager is used to control the two phase commitment operations. For more information on JOTM, see http://jotm.objectweb.org .

RIB components

The SeeBeyond components listed above build and process RIB messages. This section lists the subsystems deployed within these components and within other Retek application software. Each RIB component has a dedicated task and is generally specific to one Message Family.

Old and New Stored Procedure Interfaces

In previous releases, all adapters used the same interface structure to the database. The main facets of this design involved the use of Oracle CLOBS (Character Large Object Binaries) as the means to pass information to and from an Oracle Stored Procedure. The stored procedure was responsible for encoding and parsing the message payload.

For those interfaces requiring a high-level of performance, this design has been modified such that XML creation and parsing is performed in the SeeBeyond e*Way adapter. The means to communicate data to/from the stored procedure is performed via the use of Oracle Objects. These objects provide a hierarchical container to store the XML and map one-to-one with all attributes and elements found in the payload of a RIB message. There are also other changes to these interfaces concerning the number and types of the parameters.

Additionally, the RIB has begun to interface with Retek applications developed on the J2EE platform. For this platform, the interface to the RIB is via a Message Driven Bean (MDB) for subscribers and by using an Enterprise Java Bean (EJB) to publish messages to the RIB.

RIB Database Objects

As mentioned above, some adapters and application interfaces have been modified to use Oracle Objects to pass information to and from the stored procedure. All of the Oracle Objects used to pass payload information are created under the same base object, RIB_OBJECT. In other words, these payload objects extend RIB_OBJECT or inherit from RIB_OBJECT. Because of this, they are generically known as RIB Objects.

RIB Objects are used as both input and output parameters to the GETNXT() and CONSUME() stored procedures. Because Oracle Objects are polymorphic, a single stored procedure may accept or produce different RIB Object types, depending on the desired message to be published or consumed.

One aspect of RIB Objects is that they are hierarchical in nature. Each RIB Object corresponds to the DTD that defines the RIB Message payload. Oracle objects do not provide support for optional attributes or elements defined as a “CHOICE”, so a RIB Object will contain all possible attributes or elements contained in a DTD.

RIB Objects use nested tables and nested objects to provide the hierarchical container. The determination of whether a nested table of RIB Objects is used is determined on the cardinality of the XML sub-node. If the sub-node has a cardinality of zero or one possible instantiations, then a nested RIB Object is used. If the sub-node has a cardinality of zero or many, a nested table of RIB Objects is used.

Database Schema Owner Requirements

The ownership of a RIB Object is critical to the correct functioning of the RIB. The owner of a RIB Object must be the same as the owner of the packages in which these Database Objects are used. If the application is installed under a different Oracle user-id than the RIB uses, then the owner of the RIB Objects must be fully specified by the RIB adapter. When this scenario is present, the owner of the package containing the GETNXT() or CONSUME() stored procedure is determined and the assumption is that this user-id also owns the RIB Objects as well.

The implication of this is that when installing a Retek application under a different user-id, synonyms for all of the packages containing GETNXT() and CONSUME() must also be present for the RIB user-id. Furthermore, these appropriate privileges for accessing the RIB Objects and executing the stored procedures must also be granted to the RIB user-id. Most often, the two privileges needed for a separate RIB user-id above those normally granted are 'CREATE ANY TYPE' and 'EXECUTE ANY TYPE'.

RIB_XML database package

In previous releases, application specific Stored Procedures created or parsed XML strings stored in CLOBS. Retek developed the RIB_XML PL/SQL Package to contain utility and helper procedures for this.

Message validation: The RIB_XML package can perform message payload validation against a Document Type Definition (DTD). This DTD is stored as a CLOB within the database. If the publishing or subscribing application requests validation, the RIB_XML package API contains parameters to extract the DTD from rib_doctypes table, parse the DTD and then validate the message payload using the DTD.

The `rib_doctypes table` stores the DTD as a CLOB and associates the CLOB with a message name. This table must be accessible within the user ID used to create or consume RIB messages. Loading the rib_doctypes table may be performed using the *DocTypeInserter* java application.

RIB_SXW database package

Another Oracle package has been developed for creating XML payloads, the RIB_SXW package. This package provides no validation facilities, but better performance than RIB_XML. It also does not contain any parsing functions.

This package also contains restrictions in how a message may be created, such as fully populating an XML element with fields and sub-elements before moving to another node on the XML tree.

RIB_SETTINGS and RIB_TYPE_SETTINGS

PL/SQL stored procedures may use two tables to refine their behavior: RIB_SETTINGS and RIB_TYPE_SETTINGS.

The columns in the RIB_SETTINGS table describe, on a per Message Family basis:

- The number of threads to use when publishing. This is used by database triggers for determining the thread value to use for scalability purposes. Not all application triggers will use this value, but those that do (typically RMS interface points) will also implement and verify that the RIB adapter also is configured to use the same value.
- The maximum number of details to publish within a create, update, or delete message. Retek applications typically do not have a limit to the number of details a specific business object can have. Hence, a Purchase Order may be created containing tens of thousands of detail lines – each line a specific item/location combination. A single "PO Create" message containing 30,000 or so lines will require a vast amount of resident memory to parse. This column limits the "PO Create" and subsequent "PO Detail Add" messages to a set number of details.
- The number of minutes that a publishing application may wait before publishing "incomplete" business object create messages. This becomes important for business object publication that are dependent on manual processes. The purpose of this is to bound the latency between an actual business event and the publication of a message, when the message publication is delayed. For example, recording items received at a warehouse within a specific shipment may be performed by employees using hand scanners. For performance reasons, aggregating all of these item receipts into a single RIB Message is desired. However, these employees may be interrupted by a variety of disturbances (lunch, quitting time, a higher priority shipment) and the complete shipment may not be scanned for some time. In this case, the MINUTES_TIME_LAG column insures that all recorded items have a known maximum latency between the scanning operation and the message publication. Note: not all applications make use of this parameter.

The columns in the RIB_TYPE_SETTINGS table describe, on a per Message Family / message type combination whether informational and debugging log entries should be output using the DBMS_OUTPUT Oracle package and/or written to a log file. These entries are not used by all applications – and may in fact be only used by Retek Distribution Management interfaces. Typically, they should only be used to debug performance or bugs found within a application.

Application message publishing triggers using CLOBs

Oracle Forms based or PL/SQL based RIB applications use triggers to initiate the message publishing process. These triggers are RIB specific and should be enabled only when an enterprise is using the RIB for integrating its applications. These triggers are fired when a specific database table is modified. There are two types of these triggers used by the RIB: those that create a CLOB to store the XML data associated with the triggering business event and those that do not.

CLOB creation triggers assume that the application is responsible for the modified data. The trigger retrieves all of pertinent information to create a specific type of message and inserts it into a staging table using an application specific Message Family Manager (MFM) API.

The message information is usually stored as an XML string and is known as the RIB message "payload". The payload is contained in an Oracle Character Large Object Binary (CLOB). The database table that holds the payload data must also maintain the following:

- The order that messages are created
- The CLOB containing the "payload" XML
- Any routing or filtering key values
- The message type associated with the business event that created the message. The message type is specific to the Message Family and a single business event may produce multiple messages of differing types within different families.

By storing all of the data within the same transaction as the business event, all RIB messages are considered as being "published" synchronously with the business event – even though the message has not been processed by any EAI system deployed component.

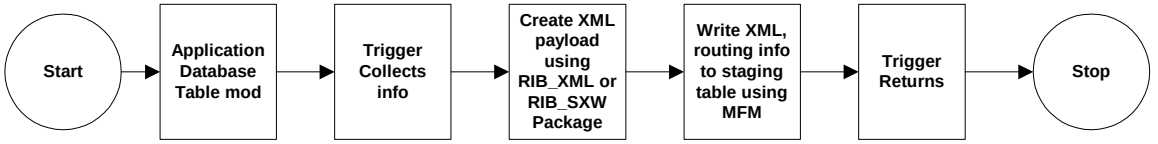


Figure 3.1 Trigger Processing -- XML CLOB

Figure 3.1 displays the application trigger processing. The following steps are followed:

1 An insert/update/delete operation on a table causes a RIB application trigger to be executed. The trigger was installed and enabled as part of the RIB installation.

2 The trigger collects any information it needs to continue. This may involve additional database operations.

3 The trigger leverages either the RIB_XML or RIB_SXW package to build the XML payload for this message type. An Oracle CLOB is created to store the XML payload.

4 The trigger calls the Message Family Manager package to store the message into a staging table. The specific API that is called is the ADDTOQ() procedure.

5 The trigger returns.

Note: CLOB creation triggers insure that all available data needed for creating the final XML is available within the same transaction as the triggering event. Because of this, there are no windows of opportunity for data to become out of sync with the published message.

Application message publishing triggers using RIB Objects

One problem with CLOB based triggers is the per-detail overhead required. Part of the overhead involves the performance characteristics creating a CLOB. Furthermore, if a Purchase Order contains thousands of detail lines, then the detail table trigger needs to be fired thousands of times. Compounding this problem is the fact that many times a Retek application will fire the same trigger multiple times within a single transaction for the same data row. Because of problems with triggers maintaining context information, this implies that the same logic is implemented multiple times. This leads to performance problems either to maintain the “correct” version of the business object in the MFM staging table or requiring extra messages to be published.

For high-volume interfaces, CLOB creation triggers are not used. Instead, detail table triggers are implemented that perform a minimum amount of processing. Many times these triggers simply check to see if the business object containing the detail has been published or does not require an approval to be performed. If so, the data required to create a “Detail Add”, “Detail Update” , or “Detail Delete” message is inserted into a staging table. Because XML strings are not created and CLOBS are not used, these operations are very efficient. If the business object requires an approval operation to be performed before it can be published, it is assumed that the correct data will be made available when the approval takes place.

When a message is ready for publication, the Message Family Manager GETNXT() Stored Procedure examines its staging tables and creates the appropriate RIB Object for publication. In many cases, these staging tables contain columns that are themselves declared a specific type of RIB Object. Once the complete RIB Object is ready, the GETNXT() Stored Procedure returns this to the adapter, which then converts the information into an XML string. This XML string is then placed into a RIB Message payload

Note that one implication of these triggers is that multiple staging tables may be needed for a single Message Family: One to hold “Header” level information and one for detail level information. Furthermore, the lifecycle of the “Header” table must map to the lifecycle of application business object itself – header information must be maintained for all periods of time that operations are valid against that business object. In other words, the header information must be kept until the business object is either deleted or considered “closed”.

RIB Objects: an in-depth view

RIB Objects use the Oracle Objects type introduced into the Oracle Database in the Oracle 8i release. This is an object-based technology that allows a developer to create database types that are hierarchical in nature and can leverage type inheritance and polymorphism. Furthermore, methods may be defined for each type similar to C++ and Java objects.

RIB Objects all inherit from a single base object type, RIB_OBJECT. A new RIB object type is created for each node on a message's XML DTD. An example of a script used to create a simple, flat RIB Object is seen below:

```
CREATE OR REPLACE TYPE RIB_FrtTermDesc_REC UNDER RIB_OBJECT (
   freight_terms  VARCHAR2(30),
   term_desc   VARCHAR2(240),
   enabled_flag   VARCHAR2(1),
   start_date_active DATE,
   end_date_active   DATE,
   overriding member procedure appendNodeValues( i_prefix in
varchar2)
);
/
CREATE OR REPLACE TYPE BODY RIB_FrtTermDesc_REC AS
overriding member procedure appendNodeValues( i_prefix in varchar2)
IS
tbl RIB_object_tbl;
l_new_pre varchar2(4000);
begin
   rib_obj_util.g_RIB_element_values(i_prefix||'freight_terms') :=
freight_terms;
   rib_obj_util.g_RIB_element_values(i_prefix||'term_desc') :=
term_desc;
   rib_obj_util.g_RIB_element_values(i_prefix||'enabled_flag') :=
enabled_flag;
   rib_obj_util.g_RIB_element_values(i_prefix||'start_date_active')
:=  TO_CHAR( start_date_active, RIB_obj_util.g_date_format)
;
   rib_obj_util.g_RIB_element_values(i_prefix||'end_date_active') :=
TO_CHAR( end_date_active, RIB_obj_util.g_date_format)
;
END AppendNodeValues;
END;
/
```

The first block of code creates the type specification. This defines the attributes stored by the RIB_OBJECT and declares that this object type inherits from the RIB_OBJECT type. The second block of code creates the type body containing the method, appendNodeValues(). This method is used only for debugging purposes.

For hierarchical structures, the "leaf" or "child" RIB Objects must be created before the "trunk" or "parent" objects. The script below creates a hierarchical structure that contains a single header and many details:

```
CREATE TYPE RIB_Detail_REC UNDER RIB_OBJECT (
varchar_detail VARCHAR2(20),
        number_detail     NUMBER(4,0),
        date_detail       DATE,
   overriding member procedure appendNodeValues( i_prefix in
varchar2)
);
/
CREATE TYPE BODY RIB_Detail_REC AS
overriding member procedure appendNodeValues( i_prefix in varchar2)
IS
tbl RIB_object_tbl;
l_new_pre varchar2(4000);
begin
   rib_obj_util.g_RIB_element_values(i_prefix||'varchar_detail') :=
varchar_detail;
   rib_obj_util.g_RIB_element_values(i_prefix||'number_detail') :=
number_detail;
   rib_obj_util.g_RIB_element_values(i_prefix||'date_detail') :=
TO_CHAR( date_detail, RIB_obj_util.g_date_format);


END AppendNodeValues;
END;
/
CREATE TYPE RIB_Detail_TBL AS TABLE OF RIB_Detail_REC;
/


CREATE TYPE RIB_Header_REC UNDER RIB_OBJECT (
   Varchar_header VARCHAR2(10),
Number_header  NUMBER(12,4),
Date_header       DATE,
Detail_tbl           RIB_Detail_TBL,
   overriding member procedure appendNodeValues( i_prefix in
varchar2)
```

```
);
/
CREATE TYPE BODY RIB_VendorHdrDesc_REC AS
overriding member procedure appendNodeValues( i_prefix in varchar2)
IS
tbl RIB_object_tbl;
l_new_pre varchar2(4000);
begin
   rib_obj_util.g_RIB_element_values(i_prefix||'varchar_header') :=
varchar_header;
   rib_obj_util.g_RIB_element_values(i_prefix||'number_header') :=
number_header;
   rib_obj_util.g_RIB_element_values(i_prefix||'date_header') :=
TO_CHAR( end_date_active, RIB_obj_util.g_date_format);


   l_new_pre :=i_prefix||'detail_TBL.';
   FOR INDX IN detail_TBL.FIRST()..detail_TBL.LAST() LOOP
      detail_TBL(indx).appendNodeValues(
i_prefix||indx||'detail_TBL.');
      RIB_obj_util.g_RIB_table_names(l_new_pre) := indx;
   END LOOP;


END AppendNodeValues;
END;
/
```

In the hierarchical example, three types are created: RIB_detail_REC, RIB_Detail_TBL, and RIB_Header_REC. The RIB_header_REC type contains a table of Details. Since the size of this table is unbounded, it must be declared as a nested table type (RIB_Detail_TBL). The resultant object types created have a one-to-one mapping to the following DTD:

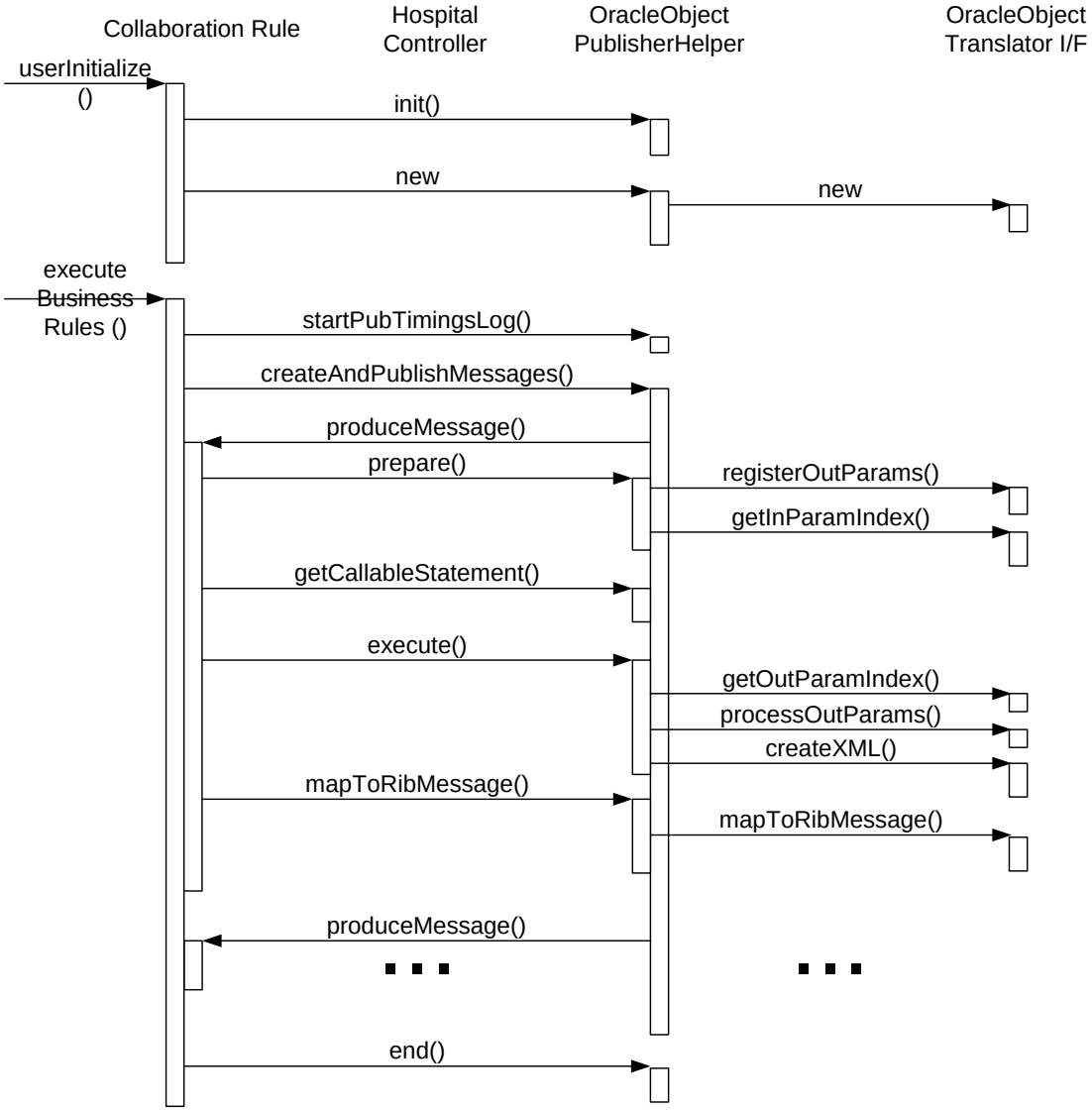
```
<!ELEMENT header (
  varchar_header
, number_header
, date_header
, details+

)>

<!ELEMENT details (
  varchar_detail
, number_detail
, date_detail

)>

<!ENTITY % varchar2 "(#PCDATA)">

<!ENTITY % number "(#PCDATA)">

<!ELEMENT year %number;>

<!ELEMENT month %number;>

<!ELEMENT day %number;>

<!ELEMENT hour %number;>

<!ELEMENT minute %number;>

<!ELEMENT second %number;>

<!ENTITY % date "( year, month, day, ( hour, minute, second )? )">

<!ELEMENT varchar_header %varchar2; >
<!ELEMENT number_header %number; >
<!ELEMENT date_header %date; >

<!ELEMENT varchar_detail %varchar2; >
<!ELEMENT number_detail %number; >
<!ELEMENT date_detail %date; >
```

Note: Dropping Oracle Object types must use the “FORCE” keyword if there any types or tables that are dependent on that type. Once an Oracle Type is dropped, all dependent types and table columns are marked has invalid and must be recompiled or re-created.

RIB Object to XML Translation

Parsing the XML to create a RIB Object and creating XML from a RIB Object are performed using different Java classes. A basic overview of these techniques is listed here.

For publishing, the adapter uses a class that implements the com.retek.rib.collab.OracleObjectPublisherTranslator interface. A class that implements this interface is known as a "PubTrans" class. A PubTrans class is coupled to the DTD of the resultant XML and the structure of the RIB Object. As such, each Message Family publisher must have its own PubTrans class. For convenience, PubTrans classes also extend the class com.retek.rib.collab.AbstractGetNextPubHelper. Combined, these classes collaborate with the OracleObjectPublisherHelper class to call the GETNXT() stored procedure and create the XML payload. A diagram of how these classes interact follows.



In the diagram above, each collaboration rule creates and contains a single OracleObjectPublisherHelper object. The OracleObjectPublisherHelper creates a PubTrans object, since the class name of the PubTrans class is supplied as a constructor parameter. Another parameter to the constructor is the collaboration rule itself, which implements the com.retek.rib.collab.PublishMessageIface interface. When the collaboration rule executes, it calls createAndPublishMessages() which performs a callback on the produceMessage() method for each ribMessage node to add into the message. OracleObjectPublisherHelper publishes the message to the JMS topic and returns a status to the collaboration rule. Finally, the collaboration rule calls the OracleObjectPublisherHelper.end() to clean up and log the correct timings entry for the method.

For subscribing adapters calling the CONSUME() API, translation is performed by examining the structure of the RIB Object and pulling out XML attributes with the same names as the RIB Object attributes. The process followed is:

1 During the collaboration rule initialization, a mapping is created that associates the correct Oracle RIB Object type name, the correct CONSUME() parameter list and a message type for all message types known by the collaboration rule.

2 The mapping process will involve examining and storing the Oracle RIB Object structure definition. The characteristics of each RIB Object attribute – whether it is a scalar value, a date, a nested RIB Object or a table of nested RIB Objects – is also stored.

3 A SAX parser is created to parse the XML payload. A parameter to the handler for the parser is the OracleObjectDescriptor mappings.

4 For each RibMessage node payload, the SAX Parser is invoked and the appropriate JDBC driver STRUCT object is created. Then, the CONSUME() method is called.

Non-trigger PL/SQL publishing

Some applications may not use triggers to start the publishing process. Some alternatives used are:

- Using an insert into the MFM staging table directly from Oracle Forms. In this case, the logic to create the CLOB and insert it into the MFM staging table is found in a stored procedure referenced directly by the Oracle Forms based application.
- Using “upload” tables to stage the information until ready to publish. In this scenario, the message is not bound to the XML format until the Message Family Manager GETNXT() stored procedures invoked. GETNXT() is described in the next section.
- Using a file to create the RIB Messages. This would typically be used for interfaces from external systems.
- Using a RIB Publishing EJB within the J2EE platform.
- Using a RIB Publishing Component within the ISO platform.

In first two cases above, the information contained in the message published to the bus is stored within the same transaction as the business event. The actual technique used to kick off a message’s publication is described in more detail in the Retek 10.3 Integration Guide.

Message Family Manager API

Each PL/SQL based application uses a Message Family Manager (MFM) specific API for publishing all messages within a specific Message Family. This API is the interface to a stored procedure package and wrappers the staging table and additional business logic surrounding the message publication. A single application is responsible for publishing all messages within a single MFM.

Because the same application can publish multiple Message Families, it could use multiple MFM specific packages, one per MFM.

There are two procedures typically included in an MFM package:

ADDTOQ()

The purpose of ADDTOQ() is to store message state, routing / filtering keys, message type, XML Payload, and other information needed to create a RIB Message. This procedure has the following format for its parameter footprint for CLOB creation based publishers:

```
PROCEDURE ADDTOQ( O_status_code     OUT   VARCHAR2,
                                O_error_text      OUT   VARCHAR2,
                                I_message_type    IN       VARCHAR2,
                                I_message         IN       CLOB,
                                I_msg_1        IN       tbl.msg_spec_1%TYPE,
                                …
);
```

where

O_status_code	Denotes the status of the call. The value of this is found in the RIB_CODES package. Possible values include: MFM_FATAL_ERROR – cannot insert a message due to an error. MFM_SUCCESS – successful message insertion.
O_error_text	This is text associated with an error or warning occurring in the call to ADDTOQ.
I_message_type	Type of the message payload. A specific type is associated with one or more business events. This type is a further subdivision of the Message Family.
I_message	The message payload formatted as an XML string.
I_msg_1	A Message Family specific facility type, key, or other information that is supposed to be present in the message envelope. This is an optional parameter and may not be present. The type of this parameter is specific to the Message Family.
…	Additional optional parameters. These are dependent on the Message Family in use.

For RIB Object based publishing, the ADDTOQ() is dependent of the Message Family, the RIB Object required, and the trigger used to publish. The parameter list is thus extremely specific to the business object or business detail involved. An example of a RIB Object ADDTOQ() is seen below for the RMSMFM_ORDERS package:

```
ADDTOQ(O_error_message OUT VARCHAR2,
    I_message_type     IN  ORDER_MFQUEUE.MESSAGE_TYPE%TYPE,
    I_order_no         IN  ORDHEAD.ORDER_NO%TYPE,
    I_order_header_status IN ORDHEAD.STATUS%TYPE,
    I_supplier         IN  ORDHEAD.SUPPLIER%TYPE,
    I_item             IN  ORDLOC.ITEM%TYPE,
    I_location         IN  ORDLOC.LOCATION%TYPE,
    I_loc_type         IN  ORDLOC.LOC_TYPE%TYPE,
    I_physical_location IN  ORDLOC.LOC_TYPE%TYPE)
```

In this case, only the minimum amount of information is available in the API for ADDTOQ(). Additional information will be queried either within ADDTOQ() or within the GETNXT() Stored Procedure.

GETNXT()

Retrieves the record from the staging table for publication. This procedure uses the following parameter signature for CLOB creation based publishers:

```
 PROCEDURE GETNXT( O_status_code    OUT   VARCHAR2,
                                 O_error_text      OUT   VARCHAR2,
                                 O_message_type    OUT   VARCHAR2,
                                 O_message         OUT   CLOB,
                                 O_msg_1        OUT   tbl.msg_spec_1%TYPE,
                                 …
);
```

where

O_status_code	Denotes the status of the call. The value of this is found in the RIB_CODES package. There are for possible values: MFM_FATAL_ERROR – cannot retrieve a message due to an error. Publisher should exit. MFM_WARNING – the next message cannot be published because of a sequencing problem. MFM_SUCCESS – successful message retrieval. MFM_NO_MSG – no messages are waiting to be put onto the integration bus.
O_error_text	Text associated with an error or warning.
O_message_type	Type of the message payload. A specific type is associated with one or more business events.
O_message	The message payload formatted as an XML string.
O_msg_1	A Message Family specific facility type, key, or other information that is supposed to be present in the message envelope. The Type of this parameter is specific to the Message Family.
…	Additional optional Message Family specific parameters.

For RIB Object publishers, the minimum signature of a Stored Procedure is shown below. Note that for a given GETNXT(), there may be additional parameters. The values of these parameters are typically specified in the RIB Properties file.

```
PROCEDURE GETNXT( O_status_code      OUT  VARCHAR2,
                  O_error_text       OUT  VARCHAR2,
                  O_message_type     OUT  VARCHAR2,
                  O_message          OUT  RIB_OBJECT,
                  O_bus_obj_id       OUT  RIB_BUSOBJID_TBL,
                  O_routing_info     OUT  RIB_ROUTINGINFO_TBL,
                  I_num_threads   IN      NUMBER DEFAULT 1,
               I_thread_val    IN      NUMBER DEFAULT 1)
```

where

`O_status_code` Denotes the status of the call. The value of this is found in the RIB_CODES package. There are for possible values:

MFM_FATAL_ERROR – cannot retrieve a message due to an error. Publisher should exit.

MFM_SUCCESS – successful message retrieval.

MFM_NO_MSG – no messages are waiting to be put onto the integration bus.

MFM_HOSPITAL – put the message into the error hospital

`O_error_text` Text associated with an error or warning.

`O_message_type` Type of the message payload. A specific type is associated with one or more business events.

`O_message` The message payload as a RIB Object. The actual type used is dependent on the Message Family and Message Type for this RIB Message. Note that many Message Types may use the same RIB Object to convey data.

`O_bus_obj_id` An identification of the ID of the business object associated with the message. This ID is unique to the Message Family. The Business Object ID may be a composite entity – for example a combination of a ASN and a distribution center ID. RIB sequencing automatically insures that all messages for a specific Business Object ID are delivered in the correct order.

`O_routing_info` Certain Message Family messages require routing operations by TAFR adapters. The information used to route these messages is found in the RIB Message envelope. In the CLOB creation interface, each Message Family had its own set of specific parameters it returned to populate these fields. In the RIB Object creation interface, the O_routing_info parameter contains this information.

`I_num_threads`	The total number of threads used in publishing. This value comes from the rib.properties file and should match the same entry in the RIB_SETTINGS table.
`I_thread_val`	An identification that this call is made for publishing messages specific to a specific thread. This value will be attached as the “threadValue” property associated with the published RIB Message.

Publishing application adapters using PL/SQL interfaces

PL/SQL based applications publish messages using at least two separate database transactions, as seen in Figure 3-2. The first transaction consists of the application specific insert/update/delete operations that perform some business functionality. These operations occur independently of the RIB. However, when the RIB is active, additional triggers are enabled on these tables that insert information into staging tables for later publication. This data may be a CLOB, a specific RIB Object sub-type, or as stored within a standard SQL type.

The second transaction is controlled by the publishing adapter. A RIB Publishing Adapter polls the staging table by calling another routine in the MFM called "GETNXT()". This type of operation is known as a "Pull", since the adapter pulls the data from the database. The MFM "GETNXT()" procedure may contain simple or complex logic that is specific to the Message Types published. For example, a simple "Create Vendor" message may involve merely selecting and then deleting a single record from the vendor staging table. On the other hand, a "Create Purchase Order" message requires fairly complex logic to create because of the business process dependencies. Many changes may be made to a PO before it is approved.

When the call to the MFM GETNXT() returns the data to the publishing adapter, a RIB Message is created from the payload (and other) GETNXT parameters. This message is then published to a Java Message Service (JMS) Topic (sometimes called a "RIB Queue").

Note: In the Java Message Service nomenclature, one puts a message onto a JMS "Topic" for Pub/Sub operations. One puts a message onto a JMS "Queue" when only a single subscriber will ever receive the message. The RIB assumes that any published message may have multiple subscribers and hence only uses JMS topics.

In the 10.3 release, Retek implemented the ability to call the GETNXT() Stored Procedure multiple times. When message data is returned, the associated XML String is created and placed within the "<messageData>" tag. (See Chapter 2 for more information on the message structure). "<messageData>" is a sub-element of "<ribMessage>".

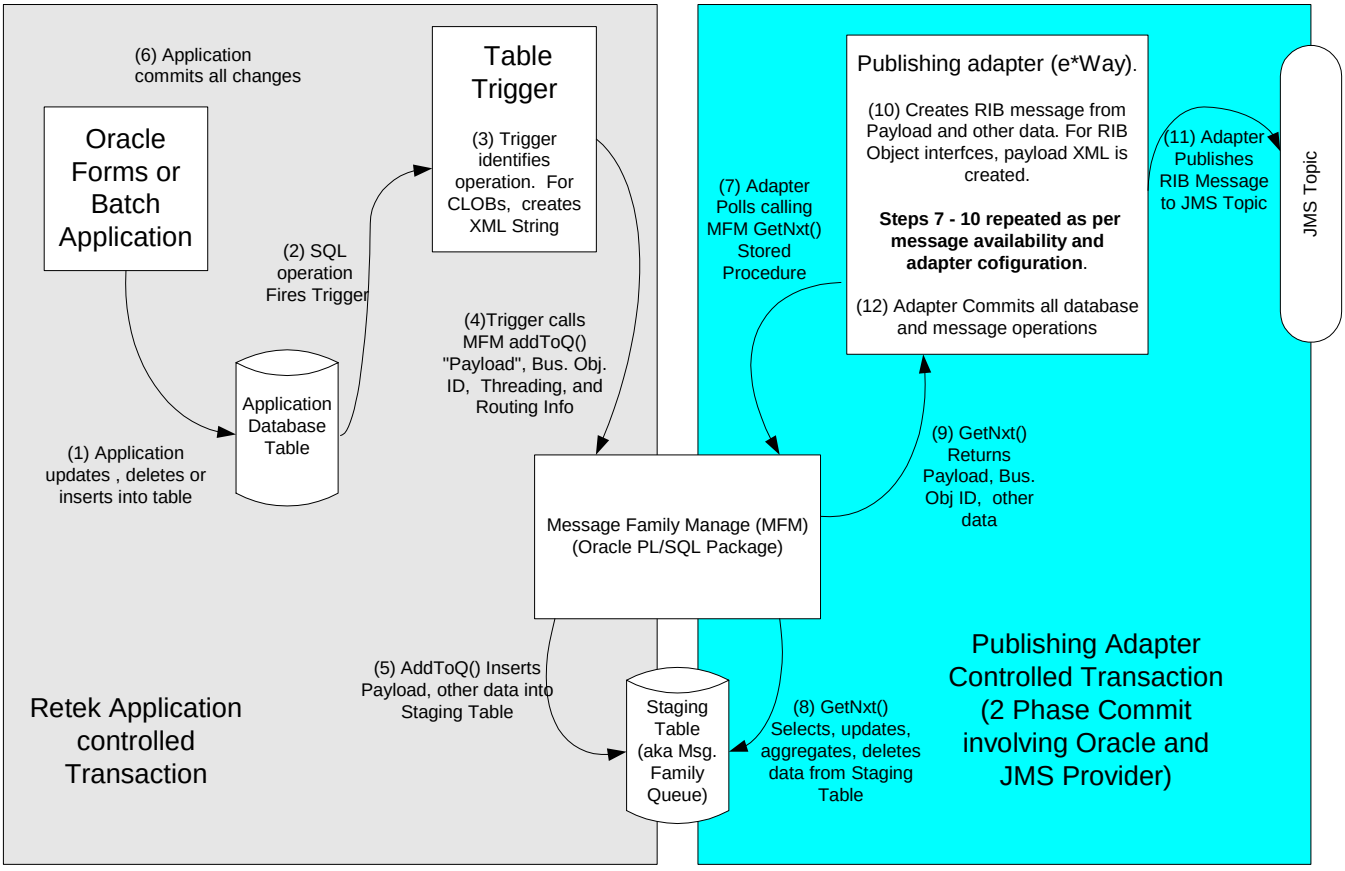
In each published message, the <RibMessages> tag wrappers one or more <ribMessage> tags. Under normal circumstances, GETNXT() is called until either a configured maximum number of times or until GETNXT() returns a "No Data Found" status.

An XA compliant two-phase commit operation is then performed to insure that all operations on the database and the JMS Topic are performed atomically. i.e. either the data is deleted from the database and published to the JMS Topic, or neither deletion nor publication occurs.

Note: XA is a Distributed Transaction Processing specification originally developed in 1991. It is now available from "The Open Group". Copies of this standard (*C193 Distributed TP: The XA Specification* ISBN 1-872630-24-3) are available from "The Open Group's" website, http://www.opengroup.org.

As long as the GETNXT() procedure returns at least one populated <messageData> tag, the publishing adapter will immediately publish the message and repeat the process. If GETNXT() returns a "No message available" status, the publishing adapter will sleep a configured amount of time before it tries to call GETNXT() again. A rollback operation will be performed if no messages are published.

The message resides in a network queue immediately after publication. This queue provides stable storage for the message in case of a system crash occurring before all message destinations receive and process it.



Sucessful Message Publication Process

Figure 3-2

TAFR Adapter

A Transformation Address Filter/Router (TAFR) adapter is another e*Way adapter that is used to process data. It contains one or more collaborations that perform TAFR operations on all messages from a single Message Family. The specific activities performed are dependent on the needs of its subscribers.

Figure 3.3 illustrates the activities associated with a TAFR adapter. These include:

1 A message is delivered to the TAFR adapter collaboration after it has been placed onto a JMS topic. This triggers the collaboration logic.

2 The TAFR performs its needed filtering and transformation processing on the message.

3 If the message is to be routed to one or more destinations, the message contents are copied into a new SeeBeyond Event Type. This event type is specific to the destination. Hence, if an Advance Ship Notice Inbound message needs to go three different warehouses, then the full contents of the message is published to the integration bus as three different events using three different event types. This allows for each of these messages to be published to different queues.

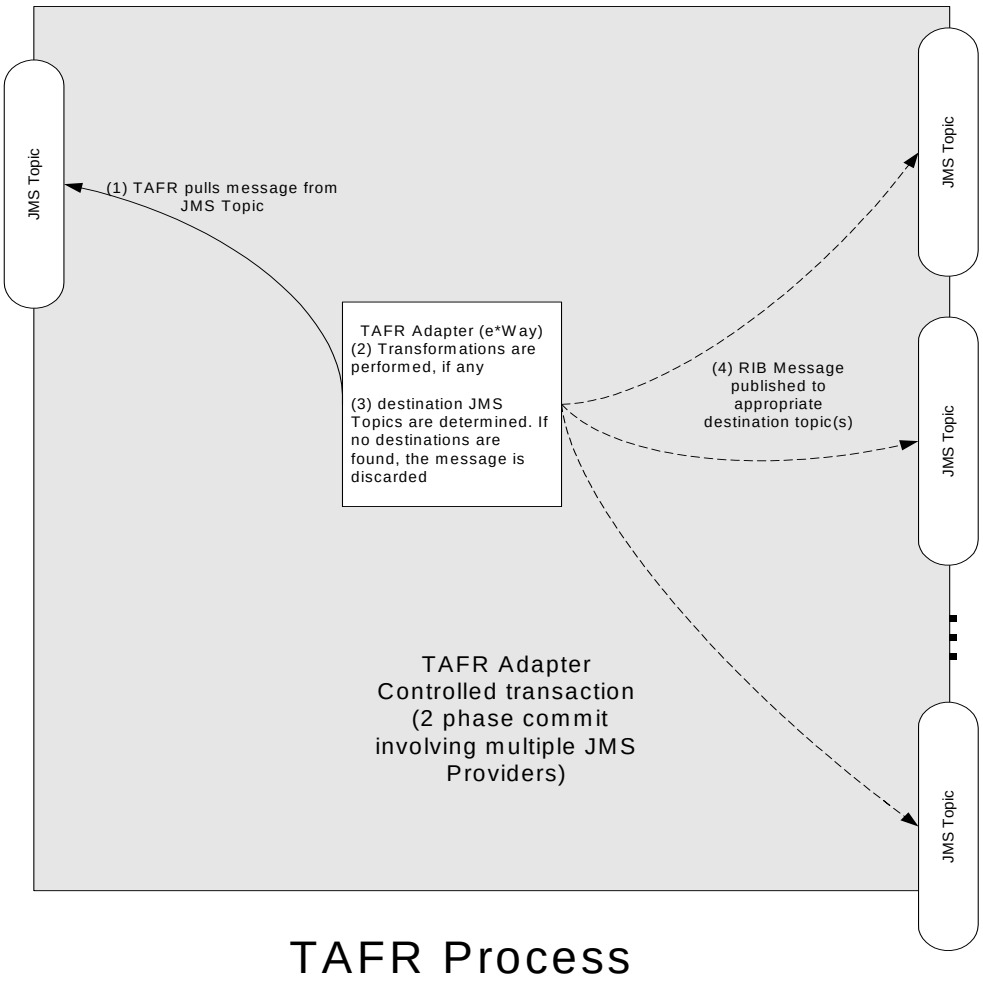


Figure 3-3

Subscribing application adapter for PL/SQL application interfaces

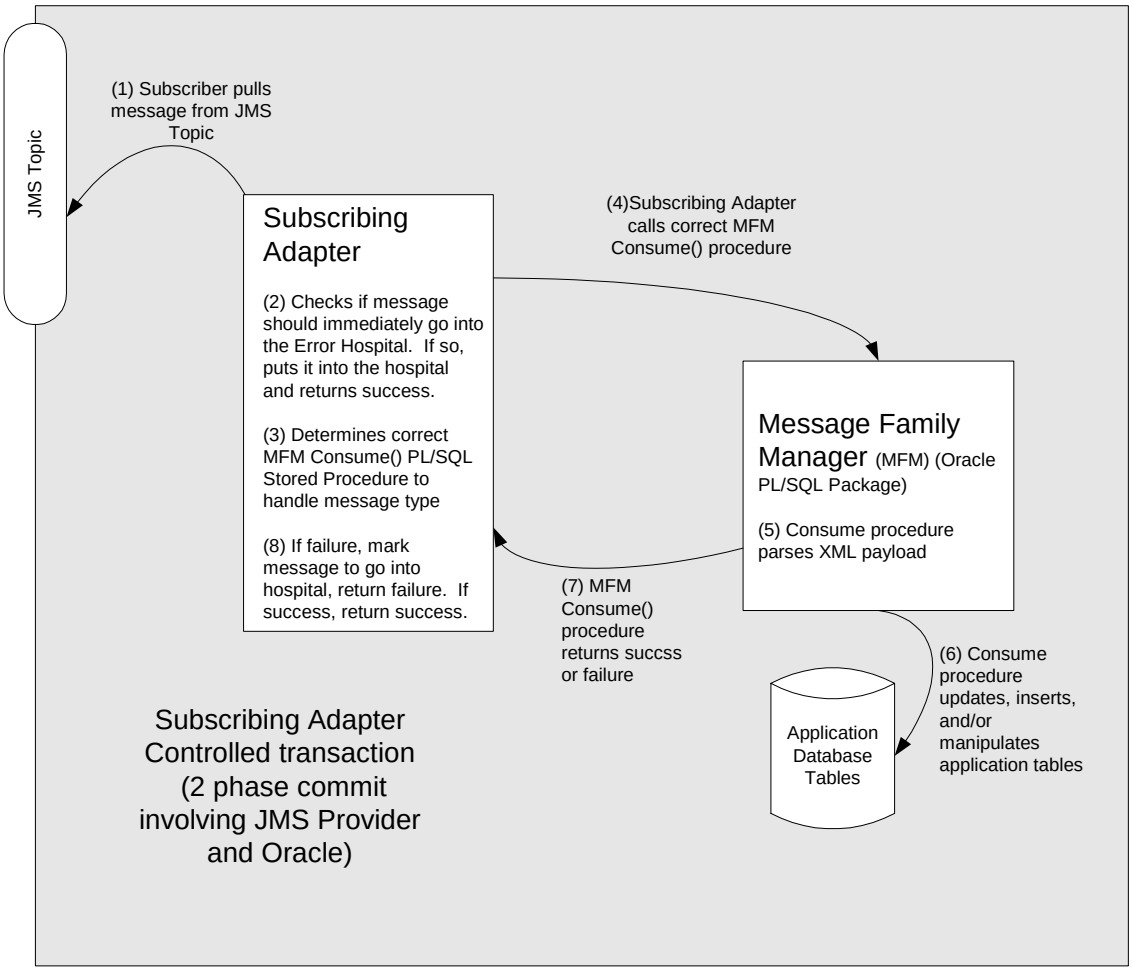
As in publishing, subscribing applications providing a PL/SQL API have two variants: one using CLOBs and one using RIB Objects. In both cases, a call is made to a Stored Procedure called CONSUME(). The purpose of this procedure is to directly update the application controlled tables with the information found in a specific RIB message type. However, for the CLOB API's, there is one specific PL/SQL Package for each separate Message Family/Message Type combination and for RIB Object API's, there is only one PL/SQL package per Message Family.

Subscribing adapters are also responsible for insuring that messages are processed in the correct sequence for a given business entity. For a specific Purchase Order, its "Create Purchase Order" message must always be processed before an update or delete message. Furthermore, all updates must be processed in the correct order to insure that two systems are correctly synchronized. But no such guarantee exists when comparing messages concerning different business entities. If no errors occur, messages are processed in a First-In, First-Out (FIFO) order. Alternatively, if an error occurs processing a message for one business object (PO #123), then other messages that apply to other business objects (PO's #124, #125…) should still be processed. Furthermore, all messages for the problem business object (PO #123) will be held in the Error Hospital.

If an error occurs during message processing a two-step process is followed: First, the subscribing adapter notes this internally (NOT in the database) and rolls back all database work associated with the message. Next, the JMS server re-sends the message to the adapter (since it has yet to be processed successfully), the adapter will now recognize this message is problematic (sick) and checks it into an Error Hospital database.

A subscribing adapter always checks the hospital database to see if there are any messages in the hospital that act on the same business entity (such as a PO) that the current message does. If so, then the adapter immediately places the current message in the hospital as well. This is to insure that all messages for a given business entity are processed in the correct order. Without manual intervention, the RIB will always process the "Sick" messages for a business object before any subsequent messages that act on the same business object.

After a message is checked into the Error Hospital, a second thread of control within the adapter extracts the message from the hospital and re-publishes it to the integration bus. The message remains in the hospital during all re-tries until the subscribing adapter successfully processes it or the maximum allowed retries is reached. The subscribing application adapter contains two collaborations for each Message Family. One collaboration is triggered to process incoming messages (the "subscriber" collaboration) and the other (the "retry" collaboration) is dedicated to re-publishing messages in the Error Hospital back to the JMS topic. Every subscriber adapter has a unique "retry" event type, which allows some adapters to retry a particular message even if others have processed it successfully.



Subscription Process for PL/SQL Interfaces

Figure3-4

Figure 3.4 illustrates the processing involved for these messages:

1 The appropriate collaboration is triggered by a message from a JMS provider. This message may arrive on the JMS topic from the Error Hospital, from a publishing adapter, or from a TAFR adapter.

2 The Error Hospital Java code is called to see if this message should immediately be placed into the Error Hospital. This logic will check

   a To see if any previously processed messages for the same business entity is in the hospital. If so, then this message needs to be put into the Error Hospital to preserve message sequencing.

   b If this is the second time this message was processed because the stored procedure returned an error the first time. If so, then the expectation is that the message needs to wait a while before it is retried. The message is placed into the Error Hospital to allow other messages to flow through during this time.

   If the message is placed into the Error Hospital in this step, the database work is committed and the message is removed from the JMS topic. Steps 3-6 are not executed.

3 The correct Message Family Manager stored procedure is called. The specific stored procedure called is based on the message type of the message.

4 The stored procedure executes the appropriate application specific logic. This may involve direct updating of application logic or simply inserting the data into staging tables.

5 If step 4 returns an error, the message is flagged as “bad” (see step 2), and the transaction will be rolled back. The message is kept on the JMS topic. The next time the message is processed, it will be put into the Error Hospital.

6 If step 4 returns success, the collaboration returns success: all database updates are committed and the message is removed from the JMS topic.

At the end of each attempt to process a message, it is found in exactly one of three locations: Still on the JMS topic (because of a stored procedure problem), in the Error Hospital, or successfully consumed by the subscribing application.

Subscribing application adapters that also publish messages

Some message processing requires database locks that reduce the scalability of the system. For example, item receipt processing may hold a lock on a shipment table or a table holding open-to-buy information. In effect, processing of these messages requires locks placed on "parent" tables. There is no problem for this when processing these receipt messages in a single thread. However, as soon as multiple threads or processing is performed, threads begin to wait on these "parent" locks and even deadlocks can occur.

There are two approaches to this problem:

1 Use threading criteria based on the "parent" record locking performed by the subscriber. I.e. publish messages flowing to different subscriber threads such that different threads will never update the same "parent" records. This requires the publisher to understand the locking used by a subscriber. One problem, however, is different subscribers to the same message may have different locking profiles. Furthermore, a single message may lock multiple "parent" records from multiple database tables with different sets of "children".

2 Move the problematic locking to another adapter. The two adapters may work either in a parallel or serial fashion. Many times, it makes business sense to first perform all of the child table processing before updating the parent table and in these cases, the PL/SQL stored procedure will return a RIB Object that will be published by the original subscriber and subscribed to by another adapter.

Subscribing application PL/SQL Stored Procedure APIs

The concept of a Message Family Manager (MFM) is also used with message subscriptions within the RIB. As in the publishing side of processing, the subscribing MFM is only concerned with the XML Payload and not the entire RIB Message XML.

All MFM packages that parse and process the payload within a RIB message have the same procedure name (CONSUME) and same basic parameter list. An example is seen below:

```
PROCEDURE CONSUME(O_status_code          IN OUT  VARCHAR2,
                  O_error_message           OUT  VARCHAR2,
                  I_message              IN OUT  CLOB);
```

where

`O_status_code` is the success/failure status of the procedure call. The values of this parameter that are standard across all subscribing packages are found in the RIB_CODES package. Currently, these include:

`SUB_FATAL_ERROR` – A fatal error was encountered processing the payload.

`SUB_XML_PARSE_ERROR` – The payload could not be parsed due to a validation error.

`SUB_SUCCESS` – The payload was processed successfully

`O_status_code` may also contain values that are application specific. These values must not conflict with those listed above. These values should be listed in the *Retek 10.3 Integration Guide.*

`O_error_message` is text associated with any error condition.

`I_message` is the payload XML text used as input to the stored procedure.

```
For RIB Object subscribing applications, the I_message parameter is
declared to be of the type RIB_OBJECT.
```

Additional parameters may be present, depending on

- the specific MFM/Message Type that is processed.
- whether the CONSUME procedure also returns a RIB Object to be published.

Also note that MFMs using CLOB based API's use multiple PL/SQL packages, one per Message Type, while RIB Object based API's use a single PL/SQL package for all Message Types within an MFM.

Error Hospital

The Error Hospital is a set of Java Classes and database tables that are designed to segregate and trigger re-processing for messages that either:

- Had some error with their initial processing.

  or

- Update the same business entity with messages already in the Error Hospital.

  As of the RIB 10.3 release, some publishers will return an 'H' status to denote a problem creating a new message for a specific business object. This status may be due to database locks being held by on-line users of an Oracle Forms application. It could also be due to some data incompatibility found in the GETNXT() procedure. In any case, whenever a publisher recognizes that a message for a business object cannot be published due to one of these conditions, the message must go into the Error Hospital.

  Of course, if a subscriber encounters any errors processing a message, it will also put messages into the Error Hospital.

Each time the message is re-processed, a record is kept of the event along with the results. The intent is to provide a means to halt processing for messages that cause errors while allowing continued processing for the "good" messages.

If a message is to be inserted into the Error Hospital because of an error during processing, it is sent to the subscribing collaboration twice. This is because subscribing collaborations are executed within the context of a distributed transaction, using the XA two-phase commit protocol. This transaction is controlled by the e*Way infrastructure: If the collaboration returns success, the message is removed and all database work committed. If the collaboration returns failure, the message never leaves the integration bus queue and the database work is rolled back.

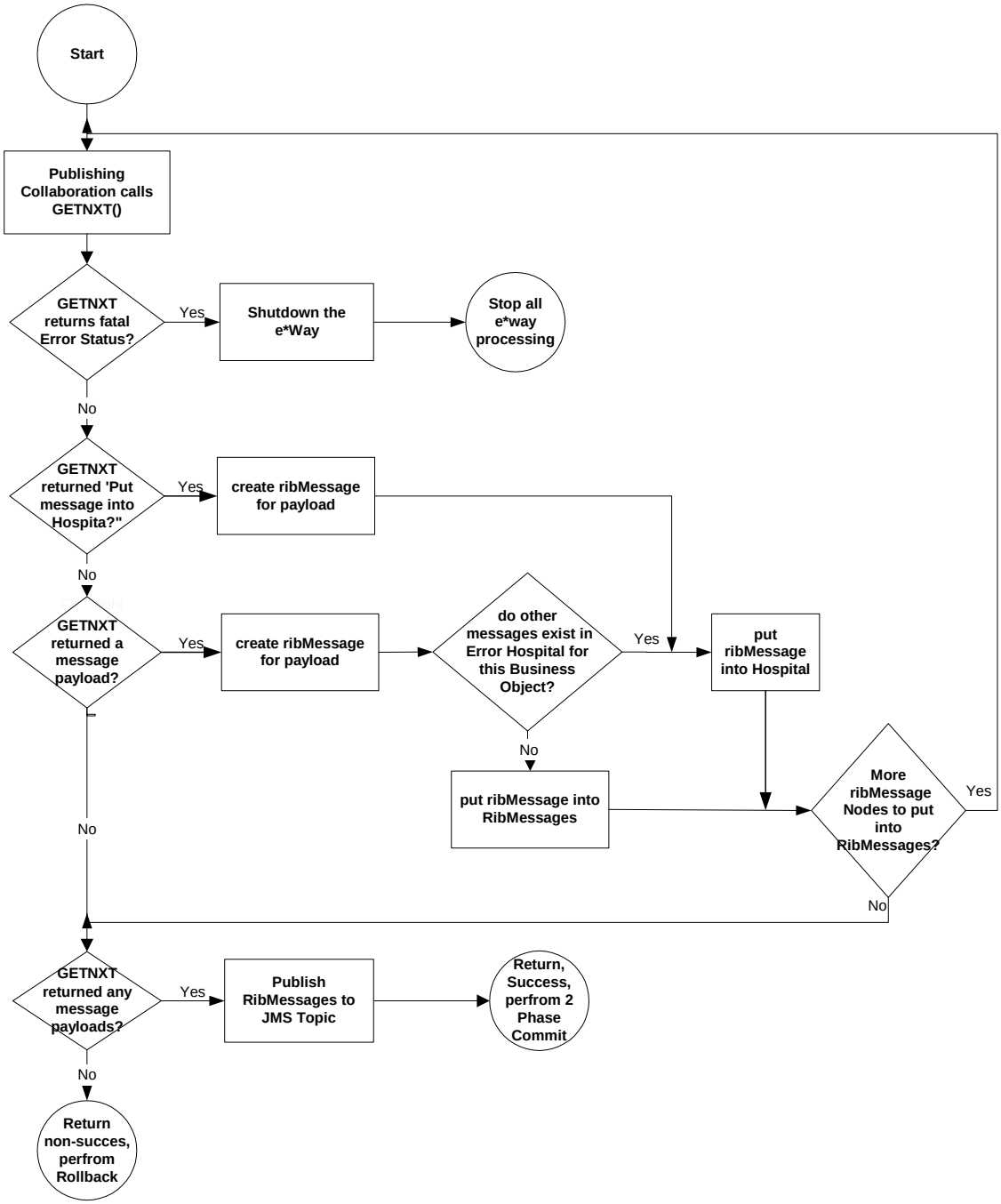
Note: The XA interface is a standard protocol between a "Transaction Manager" and a database or "Resource Manager". In a SeeBeyond e*Way, the Transaction Manager is part of the e*Way software that is involved in executing the collaboration. Note that both the JMS topic connection and the database connection must support the XA protocol. For more information regarding the XA standard, see the URL http://www.opengroup.org.

When the initial failure occurs while processing the message, the error is flagged within the Error Hospital software, the collaboration returns failure so that the database transaction is rolled back, and the message is kept on the integration bus queue. Because the message has not been successfully processed, it is re-submitted to the collaboration. This re-try will now cause the message to be inserted into the Error Hospital tables.

The Error Hospital assumes that each Message Family has a single unique ID for all business object entities its messages are associated with. This ID must be the same for the same entity across all Message Types within the Message Family. If any message for a specific business entity is admitted to the Error Hospital, then the Error Hospital will automatically insert subsequent messages for the same business object. This helps maintain correct message sequencing and guaranteed exactly once successful message processing. Otherwise, multiple update messages for a business object may be processed in an incorrect order and create incompatibilities between applications.

Pl/SQL API Publisher Processing

For a publishing adapter, the following logic is performed to publish messages to the RIB or place messages into the Publishing Error Hospital:

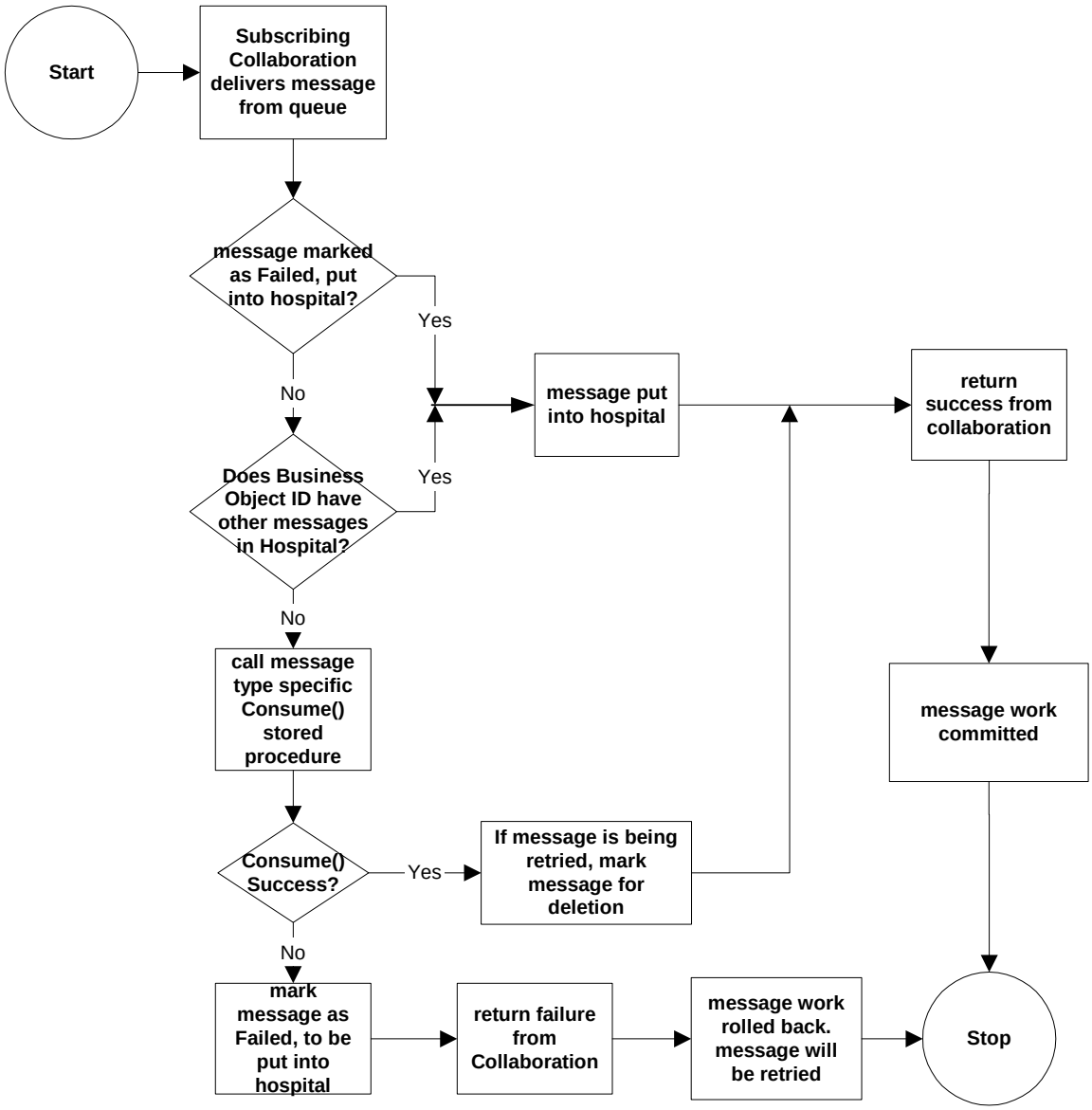


Publisher Message Processing flow using GETNXT()

Figure 3-5

Pl/SQL API Subscriber Processing

For a subscribing adapter, the following logic is performed regarding placing messages in the Error Hospital:



Subscriber Processing Logic using CONSUME()

Figure 3-6

Also associated with the Error Hospital within the subscribing adapter is a subscriber "Retry" adapter. This adapter is responsible for re-creating and re-publishing messages, which have had problems previously. There should be one subscriber "Retry" thread within the adapter per Error Hospital and JMS service provider. This thread of control is also responsible for deleting all messages marked for delete in the Error Hospital. (A Hospital Graphical User Interface application is available for manual operations on messages found in the Error Hospital.)

Messages are selected for retry based on the Business Object ID, the "Hospital ID" (a sequence number used to insure message sequencing is maintained), and whether the maximum number of automatic retries has been reached.

Error Hospital Database Tables

The following tables are used to store messages in the Error Hospital:

`rib_message` – contains the message payload, all single-field envelope information, and a concatenated string made from <id> tags. Also contains a unique hospital ID identifying this record within the hospital.

`rib_message_failure` – contains all failure information for each time the message was processed.

`rib_message_routing` – contains all of the routing element information found in the message envelope.

`rib_message_hospital_ref` – contains all of the hospital reference information found in the message envelope.

Additionally, a sequence, rib_message_seq, is used to maintain a unique "Hospital ID" associated with each message placed into the Error Hospital.

Note: The "Retry" collaboration is responsible for maintaining the "State" information for hospital records. One element of this information is whether the message has been queued to the JMS topic for re-try processing. Thus, manually deleting messages from the hospital database using SQL directly may produce severe processing problems. Similarly, deleting messages directly from the JMS provider may result in a message that is never retried again.

The RIB is supplied with a command-line and GUI interface to the Error Hospital database for administrative message control. These facilities also allow one to manually change the payload data for the next retry attempt.

Chapter 4 – RIB Message Families

This chapter presents an overview of the RIB Message Families. Each Message Family contains information specific to a related set of operations. Processing by Message Family insures that a sequence of messages for a given Business Entity (for example, a PO) is maintained throughout the message lifecycle. In the RIB 10.3 release, a single thread of processing insures this sequence. The RIB infrastructure maintains a FIFO ordering for messages on all of its queues.

A Message Family may contain multiple "Message Types". Each message type encapsulates the information specific to a business entity within one or more business events. A single business event, such as updating a Purchase Order, may involve multiple business entities, such as a line item within the Purchase Order. Furthermore, because a single business event may involve multiple business entities, the application may publish messages for this event from multiple Message Families for a single business transaction. More than one message type within a Message Family may also be created.

Messages published from *different* Message Families or messages acting on different business objects do not have the same sequential guarantees. It is possible for two Purchase Orders to be processed by a subscriber in the reverse order they were created. Many times the cause of this is due to an error or locked record discovered by the publishing adapter.

Dependencies between Message Families are more problematic. For Example, an Item must be created before it is used in a Purchase Order. If the Item publisher or subscriber is not available, then the Purchase Order may arrive at the subscriber before the Item it uses does. When it does, the PO is put into the Error Hospital. The Error Hospital retry logic then attempts to automatically correct this situation by re-publishing the PO a configurable number of times.

Event types and Message Families

Each Message Family uses a single SeeBeyond Event Type Definition to define the publishing format for all Message Types within the Message Family. Because of this, the SeeBeyond e*Gate Integrator infrastructure sees all messages from a Message Family as belonging to a single "type", known as the Event Type. The RIB message processing logic sub-divides the messages according to the message type field found in the RIB message envelope. The Event Type is the SeeBeyond ID associated with the type of the message. Event Types may use the same internal format. As such, Event Types may also be specific to how much processing has occurred on the data.

The SeeBeyond Event Type used for a Message Family may be changed if TAFR components are part of the processing stream. This is required when a single message needs to be routed to multiple destinations. In this case, each destination is associated with a distinct queue and each queue is associated with a distinct Event Type.

TAFR components may also change the Event Type messages when a mere transformation or filter operation is performed. This is done for two reasons:

1 It allows flexibility for the RIB topology. All messages may be put into the same queue on the integration bus if they have different types. For simple topologies, one can monitor the number of messages “In progress” on the RIB by looking at the statistics from a single queue.

2 It provides greater clarity when configuring a subscribing adapter or TAFR collaboration. Triggering events for a collaboration are fully specified by the Event Type and the source of the Event Type. When the source is an “upstream” collaboration, the Queue containing the event is “hidden” within the upstream collaboration’s configuration. Specifying the output event type using a different name insures that any components requiring the TAFR operation gets only TAFR processed messages.

Message Family References

An excellent resource summarizing the Message Families is the RIB_FAMILIES.pdf report supplied with each RIB installation. This document lists the available Message Families, their Message Types, the names of the DTD’s that document the message payload.

Chapter 5 – External application message interfaces

This chapter presents a brief overview of interfacing with external applications using defined RIB messages.

Direct JMS interfaces for non-Retek applications

Legacy and other applications should directly connect to the SeeBeyond JMS provider using standard JMS interfaces. For implementation specific details, see SeeBeyond e*G*ate API Developer's Guide*.

Connecting directly to the JMS provider allows an application to decouple its implementation from the Retek application. Changes made to the Retek application will not affect this interface as long as the message format remains the same.

All message publishers should publish to the JMS using JMS 'Text' messages. This insures that character encoding issues are minimized. Messages published as 'Bytes' messages could run into character encoding issues, depending on the default encodings of the Java Virtual Machines used to publish and subscribe to the message.

All message publishers must also ensure that a message is published with the JMS Message Property, *threadValue* set to an appropriate value. When only a single subscribing thread is used, the value of *threadValue* should be '1'. This is the default for all RIB adapters. When multiple threads are used, messages should be published with a value of *threadValue* that specifies the logical processing channel to use.

Furthermore, all subscribers must use selectors to insure that they do not process retried messages destined for other subscribers. Retried messages are queued onto the same topic that they originally were published to. The Error Hospital Retry publisher will set a *retryLocation* property to specify that the message is being retried and that only one specific subscriber should receive it.

A typical selector used for RIB Messages has the following form:

```
threadValue='1' and ( retryLocation is null or retryLocation =
'<adapterName>' )
```

Where <adapterName> is an identification of the subscriber. For those adapters running on the SeeBeyond eGate platform (an e*Way), it is the name of the e*Way and the name of the collaboration separated by a period. E.g. 'ewItemToRDMWH1.colItemToRDMWH'.

Character Encodings

The RIB fully supports the UTF-8 character encoding. This encoding allows for multi-byte Unicode characters to be contained in RIB messages.

At this time, Retek only fully supports UTF-8 as the Oracle database natural language. However, some implementations have used other character sets. In these cases, translation from Unicode UTF-8 to another character encoding is performed within the Oracle JDBC driver and PL/SQL interface.

RIB Messaging Paradigm concerns

The following tenets of the RIB Messaging system are of interest to external (non-Retek) publishers and subscribers:

1 During a business transaction, one or more “Create” messages may be published. These messages consist of all header and detail information for the composite entity created. External applications may require that these messages be coalesced into a single composite message.

2 Conversely, an external application may not have the same data model as the Retek application and require that a composite message be divided into multiple messages. These may need to be along the lines of a “header” and one or more “details”.

3 When a business entity is modified, a message specific to the modification is published. The message may be specific to a sub-entity. For example, if a line item is added to a Purchase Order, a PODTLCre message will be published. If multiple items will be added, multiple PODTLCre messages will be created. This means that a single database transaction may result in multiple messages within the same or multiple Message Families.

   Non-Retek subscribing applications may not associate a single message with a single database transaction. Another problem is that some non-Retek applications require a complete snapshot of the changed business object, not just a snapshot of the changed detail or header. In this case, a TAFR must be developed to create the desired information.

4 In terms of non-Retek (external) publishing applications, the application must publish using Retek’s canonical form (as specified in the Retek Integration Guide) or convert to this format. Besides converting field names or code values, this may also mean splitting up a single message into multiple messages.

5 Deletion messages may be applicable to an entire composite business entity. Different Message Types distinguish between the deletion of a sub-entity and the composite entity. For example, a Delete Supplier message will delete the supplier and all of its addresses, while a Delete Supplier Address will only delete a supplier’s address.

   Non-Retek subscribers that cannot accept a single delete message for these entities will need to have additional processing to specify the sub-entities to delete.

6 The full create/modify/delete/detail update/detail modify/detail delete Message Types are *not* available for all Message Types. Non-composite business entities do not contain “detail” operations. Some messages, such as a Stock Order Status, reflect only an adjustment to an entity that will never be deleted (or created) by the publishing application.

7 RIB published messages may require modification or transformation to satisfy the external application APIs. These modifications and transformations may involve additional database operations. For example, the complete vendor name may be needed in a message as opposed to a “vendor ID” found in the RIB message. Once the data requirements of the subscriber have been determined, the available RIB messages should be inventoried for their applicability and the specific transformations that need to be applied to them.

SeeBeyond application-specific adapters

When integrating with an existing non-Retek application, development time may be shortened considerably using a SeeBeyond e*Gate Application Adapter designed for that specific application. These application adapters are either:

- e*Ways that surface an application’s interface via a set of event type definitions: For these types of e*Ways, one must develop a set of subscribing collaborations that accept RIB messages as input events and a set of publishing collaboration that accept the application specific events.

  The subscribing collaborations convert the input RIB event into the event types associated with the non-Retek application adapter. Then the collaboration must publish the event to the “External” side of the e*Way. The “external” side then understands what API’s are used for each event type and updates the application with the correct data.

  The publishing collaborations must convert the input application specific events into one or more RIB events before publishing them. The source of these events must be the “External” side of the e*Way.

  Because of deployment limitations and performance concerns, it may be necessary to locate the message event type transformation logic within a different e*Way or BOB from the application specific e*Way. Because the conversion is already done, no transformation is needed at the application specific e*Way and “pass-through” collaborations are configured as part of the e*Way.

- A library of event type definitions or wizards used to create these ETDs: An example of this is the EDI ETD library. The purpose of these libraries is to reduce the time creating, parsing, and/or validating the message format. For example, one could use the event type definitions for EDI. In this case, the ETD library aids parsing of the EDI document and reduces the amount of development needed to convert these into messages used on the RIB.

Chapter 6 – Retek Extract, Transform, and Load

The Retek Extract, Transform and Load (RETL) is a high-performance runtime tool that is especially useful in parallel processing systems designed for high volumes of data. The design of the RETL decreases the time importing or exporting data to or from a database. An “IMPORT” operation reads from a data file and an “EXPORT” operation creates a data file.

The usage of the RETL tool should be based on desired performance and data volume. The RETL is a tool that leverages parallel processing. Although the integration bus can also be configured for parallel processing, the RETL tool set is much more flexible, and performs better. RETL is optimized specifically for high data import and export throughput – much more than a normal on-line messaging system.

The RETL software is extremely powerful and flexible. There are currently no standard event type definitions for the RETL. The relationship between the RETL and the RIB integration bus intersect only on the transfer of these files. As such, one should treat the RETL tool in the same manner as a batch job stream. The RETL may use a file as input or create a file as output. These files may be transferred like a regular batch file. However, if the RETL is used between two Retek databases, it may make sense to keep the file where it was generated and to create two batch jobs executing serially on the same host.

Note that the size of the files produced could be a concern when RETL is used. As seen in the next chapter, the easiest way to implement a batch file transfer is as a single message. However, the one-to-one association of a file to a message requires that the entire message must be read into program memory. If the file is very large, then this could consume more resources than are available, causing the file transfer e*Way to hang or error. Hence, it may be worthwhile to investigate the size of the files imported or exported via the RETL tool and, if over 100 megabytes in size, consider techniques to break the file up into smaller sizes.

Please read the 10.3 *RETL Programmer’s Guide* for more details.

Chapter 7 – Batch job integration

Retek recommends that integration to Legacy applications use JMS as the means to integrate with Retek applications. The methods to do this may include a new messaging component or may be by via a file loaded to the RIB. This section describes using the SeeBeyond "Batch" e*way to load a file to and from the RIB.

The main characteristic of a batch job is the reliance on a file as the means for input and output. In point-to-point solutions, this file is typically FTP'd between systems. To integrate with the RIB, the batch file is converted to one or multiple messages published to the integration bus.

There does not exist any pre-packaged batch integration software within the RIB 10.3 software that extracts data from the database and publishes it as a series of RIB messages versus a file. If such software existed, then this in itself would be a message-based solution (and there would still not be any pre-packaged "batch" integration). However, the SeeBeyond e*Gate Integrator infrastructure allows files to be used as sources or sinks for messages. However, an e*Way collaboration does exist that can be used to load files if these files have been already created in the correct XML format.

The RIB may be an alternative to using FTP or in conjunction with FTP file transfers. The mechanism currently used to FTP existing batch jobs may be replaced completely RIB based mechanisms.

Motivations for replacing FTP transfers

FTP is a common method for transferring files between systems. It uses a stable, well-specified protocol and mature products are available that implement it. RIB integration with batch files involves taking the file information and publishing it to the RIB. The reasons why one would want to replace an FTP transfer with this method include:

- Reduced number of FTP jobs that transfer the same file from place to place.
- With FTP, both hosts need to be available. When an adapter publishes data to a JMS topic, only the RIB and one of the hosts need to be available. Because of the distributed processing available on the RIB and the ability to move components physically within a network, there is an increased flexibility for operations personnel to perform system maintenance.
- Subscribers or publishers can move from a batch-oriented method to a message-oriented mode in an incremental fashion. After publication, file data exists as one or more messages and can be transformed, filtered, and routed as such. If the same data is needed by multiple subscribing applications, then some of the subscribers can remain relatively unchanged and still use a file as input while others can read the data as messages directly from an integration bus queue.

Transfer file data using a batch application e*Way

The first and simplest available option for using the RIB in this respect is to use the SeeBeyond e*Gate Batch application e*Way to transfer file information to and from the RIB. This e*Way can be used to copy files to or from hosts without installed e*Gate components. The Batch e*Way is fully documented in the SeeBeyond *Batch e*Way Intelligent Adapter User's Guide***.** This manual presents a brief overview of its capabilities.

Do *not* use the SeeBeyond e*Gate File e*way. This is a development tool not robust enough for deployment in a production environment.

A batch e*Way is created by creating new e*Way in the e*Gate Enterprise Manager, selecting "stcewgenericmonk.exe" as the "Executable file", and then, when creating the new configuration file, selecting the "batch" e*Way configuration template.

The Batch e*Way works in one of two modes:

1 A *fixed* configuration that publishes data to the RIB based on the presence of a file in a directory or creates/appends a file based on the presence of a message on a queue.

2 A *message* based configuration where the batch e*Way subscribes to messages that contain the specifics of the file transfer.

"Fixed" configuration

Publication of data to the RIB

A batch e*Way is configured to poll for the existence of files (either on the local system or on a remote system). Once found, the e*Way copies the files to a local temporary directory. For files found on remote systems, FTP is used to copy it to the local temporary directory. Configuration options determine the polling interval, where the file is located, file masks to determine which files to transfer, FTP parameters, whether the file should be renamed or archived after publication, and if the contents of the file should be published as a single message or if each line in the file corresponds to a single message. This is all performed in the "application" side of the e*Way.

Once a message is ready on the application side of the e*Way, the message is sent to the "collaborations" configured with the e*Way. A collaboration must be created that can handle the messages published whose source is "<external>". In the simplest case, this collaboration could merely pass through the data without modification or validation. In a more complex case, the collaboration could validate and transform the data before publishing it as an event.

If the entire file is to be published as a single message, the entire file will be read into the memory of the batch e*Way. The memory allocated for this may never be relinquished by the e*Way, depending on its scheduling. Severe problems may result when the amount of memory needed exceeds the maximum available for a single process or when the virtual memory of the machine is exhausted. Retek internal test systems successfully transferred files 100 megabytes large; your results may vary according to the specific operating system and its configuration.

Subscribing to data from the RIB

A batch e*Way is configured with a collaboration that is triggered from events (messages) published by another collaboration or are available on a JMS topic. The processing order of these events is the reverse of publication. First, the subscribing collaboration is executed and performs any needed transformations or validations. Then the message is passed over to the "application" side of the e*Way by publishing the message to the "<external>" destination.

The configuration of the application side determines the final disposition of the data. As in the publication scenario, the data stages through a temporary file and before copied to its final destination. FTP is used when the final destination is a remote system. Configuration options for this processing include the following:

- The name of the file to put the message in.
- Whether messages are appended to this file or new files are created.
- Whether the file is uniquely named via a time stamp or sequence number.
- How often new files are created (if the append mode is used) and copied.
- Pre- and post- file copy activities.
- FTP session parameters.

Import notes: When the "append messages to a file" is used, file boundaries are not necessarily maintained from the source file. One or more source files could be put into a single destination file or, if the source file was published record-by-record, half of the source file could be appended to a single destination file and half to the next. It all depends on a set of interacting configuration parameters. Furthermore, if a batch e*Way was used to publish the file using a "fixed" configuration, no intrinsic mechanism exists for communicating the name of the source file.

“Message” mode

In message mode, the batch e*Way receives an XML message detailing the file transfer details. This message contains one or more operations or commands to execute. There are two types of commands:

1 “receive” – find one or more external files and publish them to the integration bus. The message published by the e*Way is formatted using XML. It contains an identifying “return_tag” plus a “payload” tag containing the data found in the file.

2 “send” – the subscribed message is used to create or append to a destination file. The message contains a “payload” tag with the file contents. Other tags in the message detail other specifics of the file, such as the destination file name, and what to do if the destination file exists, and local/remote file copy details.

One advantage of the “message mode” FTP configuration is that “send” commands specify the name of the destination file. Hence, it is possible to maintain file names across the file transfer. However, this method requires additional development and processing.

Transferring data directly from/to a database

Another method for implementing batch transfers is to create an e*Way and a set of collaborations to read from a database table and publish the information to the RIB. This involves using the e*Gate Enterprise Manager to create the event type definitions, collaboration rules, collaborations, e*Ways and queues. This strategy replaces a batch mode of processing with a message-based mode. It directly uses new development specifically for the integration bus.

There are two strategies one can use for this development: Using connection points and developing the logic entirely within a collaboration or using one of the “Generic” SeeBeyond e*Way adapters.

Using connection points and developing the logic within a collaboration

This strategy is useful if the data is available via a simple SQL statement or with little added processing. (Actually, the wizard generates events based on table structure, SQL statement, or Stored Procedure API.) The e*Gate Enterprise Manager contains a database wizard that can generate an event corresponding to the SQL statement.

Publication: One defines an e*Way connection with a polling parameter determining how often these events will trigger the collaboration. No data or SQL statement will populate the event (message) when the collaboration triggers. The SQL statement executes as part of the collaboration rule logic and each row of any result set needs publishing as a separate event.

Subscription: One configures a collaboration that includes the defined event as an output event with a destination specified as a database connection point. The collaboration transforms the input data into the SQL specific event and then executes the SQL statement.

Note that database transaction boundaries depend on XA interface usage and an event's destination or source. If the XA interface is used, all work within each invocation of the collaboration is within a single transaction. If not, the collaboration can execute multiple transactions per single invocation. RIB collaborations typically use XA to insure "exactly once" successful message processing.

Using a "generic" e*Way application adapter

A Generic e*Way Application Adapter is useful when the business logic surrounding message creation or processing is not trivial. This series of adapters also *cannot* leverage the XA interface. There is the possibility that the same message is published or consumed multiple times.

Generic Application Adapters are specific to a programming language such as Java or C/C++. Their configuration specifies a shared library, DLL, or Jar file that contains the application logic. The functions, classes, and methods used in this logic must meet certain criteria.

These adapters have the following models:

- **Publication:** When the e*Way is instantiated (brought up) its configuration is read and the container of the application logic is attached to the e*Way. Specific initialization functions are called (as per the Generic e*Way standard application API). These functions may perform one-time activities, such as establishing a database connection. Additional functions or methods need to be implemented to inform the e*Way of lost connections or other events. Once the e*Way is initialized, it polls (according to a configured parameter) the application by calling a specific application provided function. If any data is available, the e*Way attempts to decode the returned bytes as a message in order to invoke a collaboration to process this message. All collaborations of this sort must subscribe to an event whose source is "<external>".

  The collaboration may simply pass the message through for publishing as-is or transform the event in some way. Once the message has been published successfully, a function is called on the "application" side of the e*Way to allow the application to further update state or commit updates already performed. The application polling function is called again and the process repeated. When the collaboration processing the application's message returns failure, the e*Way calls a "failure" function to allow the application to process the failure or rollback database changes.

  Between each loop there are checks to see if any the application is ready to continue or if an administrator has requested the e*Way to shut down.

- **Subscription:** In order to process incoming messages, a Generic e*Way must have at least one collaboration configured with an output event type that the application can parse. This event must also have a destination of “<external>”. Input events can come from any valid connection point or other collaboration. The collaboration processes the input event according to its own logic and publishes the output event. The e*Way presents the output event (message) as a parameter to an application-side implemented function.

Note that the application side of the e*Way is responsible for maintaining its own database connections that it uses. Any needed information can be prompted for in the e*Way configuration using modified “configuration definition files” (*.def).

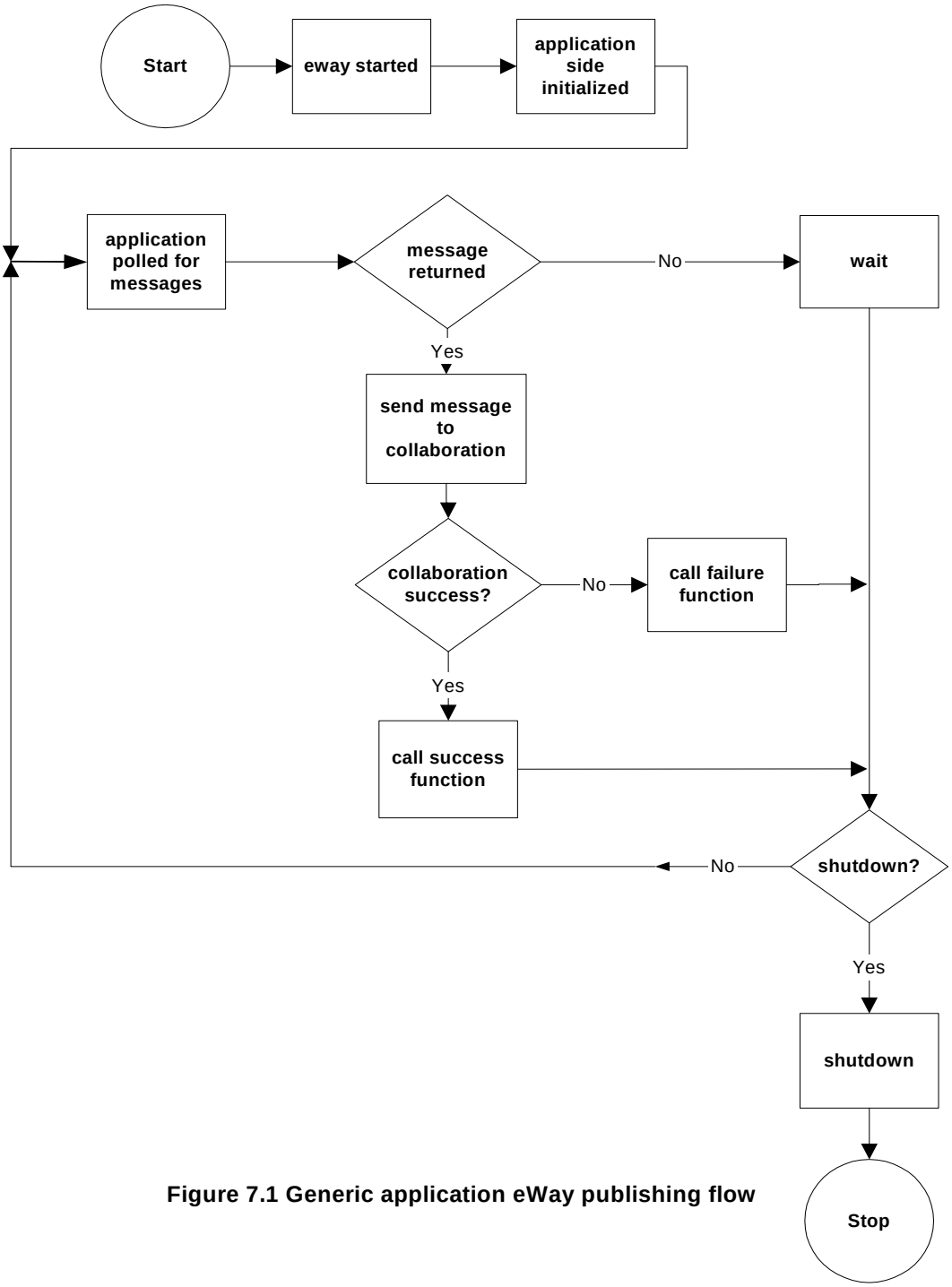


Figure 7.1 Generic application eWay publishing flow

For more information on the specifics of the Generic e*Way adapters, see the appropriate SeeBeyond manual listed below:

- Java Generic Extension Kit Developer's Guide
- C Generic e*Way Extension Kit Developer's Guide
- Generic e*Way Extension Kit (Monk enabled)

Using an application specific e*Way adapter

Application specific e*Way Adapters are built using the same paradigm as the "Generic" adapters listed above. However, these e*Ways have the "application side" of the e*Way already developed. The event types (message formats) the application can publish or parse are typically defined already (or at least an easy way to create them is available) along with the application processing logic. Hence, the main work here is to develop the correct collaborations to convert RIB events (messages) to or from this set.

There is a rich set of application specific adapters available. A complete list is available on the SeeBeyond web site, http://www.seebeyond.com.

Calling Subscribing and Publishing APIs without the use of Seebeyond

The class RmsBatch can be used to run subscribing and publishing APIs without having to use Seebeyond. The RmsBatch class will export RMS publishing data into XML files, and import XML file data into RMS subscribers. Currently, this functionality is compatible with RMS 9.

The use of the RmsBatch class is nearly identical to that of RMS batch programs. The class is run once per message family, and imports/exports all of the data in one run of the program.

Once an RmsBatch publishing run is complete, it is up to the user to send the output file to the appropriate subscribing applications. Similarly, it is up to the user to create accurate input files for the RmsBatch subscribing runs. The class accepts XML files that follow the RibMessages format.

The following message families have been configured for use between RMS 9.0 and RmsBatch:

Publishing APIs

- RMSMFM_COSTZNGRP
- RMSMFM_LOCLIST
- RMSMFM_ORGHIER
- RMSMFM_PARTNER
- RMSMFM_SEASON
- RMSMFM_STORE
- RMSMFM_SUPPLIER
- RMSMFM_WH

Subscribing APIs

- RMSSUB_ITEMLIST
- RMSSUB_XITEM
- RMSSUB_XITEMLOC
- RMSSUB_XUDA

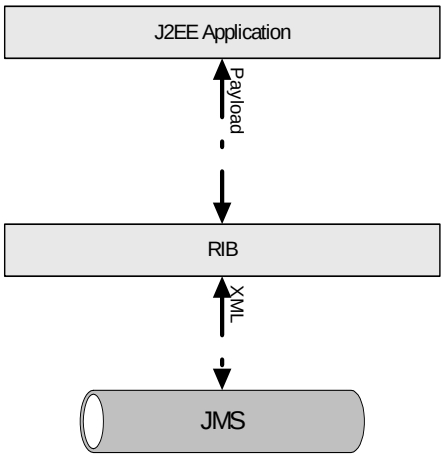
For information on how to run the RmsBatch class, consult the RIB 10.3.4 Operations Guide.

Chapter 8 – J2EE RIB Architecture

J2EE Solution Overview

This document will explain the architecture behind a J2EE application interface to the RIB. The diagram below shows the interface between the J2EE application and JMS.

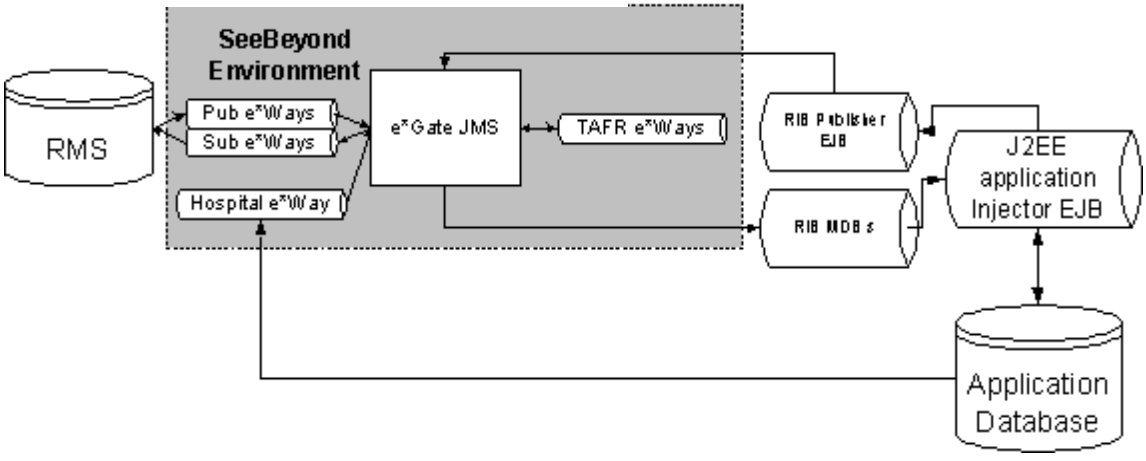
J2EE Application to JMS solution



A J2EE application interfaces with the RIB through Java Payload objects, which consist of simple Java beans that store the application data. Payload objects are converted to/from XML in the RIB through an XML Binding tool such as Castor. The RIB interfaces with JMS through the XML messages in a predefined format, based on the RibMessages DTD. See Chapter 2 for more information on the RibMessages structure.

The diagram below shows the configuration for integrating a non-J2EE application (RMS), with a J2EE application (such as RCOM).

J2EE Application to PL/SQL Application solution



RMS connects to the RIB using PL/SQL through eWays. RCOM connects to the RIB using TAFR eWays, and various Enterprise Java Beans (EJBs). Each subscribing message family has its own deployment of the RIBMessageSubscriberEJB, which is a Message-Driven Bean (MDB). The MDB listens to the JMS topics for messages, and when one appears, it processes it through its onMessage() method. All messages published from the J2EE application will use the RIBMessagePublisherEJB's publish() method to publish messages to JMS.

RIB J2EE Overview

In the J2EE environment, publishing to the RIB is performed via a deployed Enterprise Java Bean (EJB). Subscribing from the RIB is performed through deployment of a Message Driven Bean (MDB) that subscribes to a specific JMS topic with an appropriate selector. In both cases, the container manages the transaction and both the JMS and database resources are included in a two-phase commit XA compliant transaction.

The RIB's J2EE code is contained in an application EAR file named for the J2EE application that it interfaces (e.g. rib-rcom.ear). The application name is also derived in a similar manner: RIBfor<*App*> (e.g. RIBforRCOM).

Currently the application server used for deployment of J2EE applications is the WebSphere Application Server version 5. In a WAS environment, the RIBfor<*App*> application needs to be deployed on the same server instance as the J2EE application EAR, and must use the same application classloader to avoid ClassCast exceptions for shared objects between applications.

Note: This is not the case for RIBforMDM application. It can be deployed in its own WAS environment.

RIB Payload Objects

The RIB currently uses the Castor XML Binding tool for converting XML to/from Java Payload objects. The RIB performs the conversion of these objects:

- On a **subscribe**, the RIB takes the XML message from JMS (RibMessages) and converts it to a Payload object to pass on to the J2EE application (e.g. unmarshal).
- On a **publish**, the RIB takes the Payload object passed in from the J2EE application and converts it into an XML Message (RibMessages) to publish to JMS (e.g. marshal).

RIB payload objects are contained in the retek-payload.jar. This jar needs to be in the classpath for both the J2EE application and the RIB application (ie RIBfor<*App*>).

RIBMessageSubscriberEJB (MDB)

The MDB is responsible for listening to a JMS topic for messages, and upon finding a suitable message (based on the message selector), processing it through the onMessage() method. There is a different deployment of the MDB for every subscribing message family, as each MDB listens to a different topic on JMS. Also, in a multi-threaded environment, there could be a different deployment of the MDB for every thread for a specific message family.

The MDB is responsible for calling the appropriate RIB Error Hospital code and RIB Binding code for processing each XML message. The RIB Binding code is responsible for calling the J2EE application's InjectorEJB. The InjectorEJB applies the business logic to determine how the data is entered into the application database. If an InjectorException is returned from the J2EE application, the transaction will be rolled back and the XML message will be sent to the RIB Error Hospital.

Subscribing Workflow

For the subscriber process, the process is as follows. It is very similar to the SeeBeyond e*Way process.

1 The Message Driven Bean (MDB) is deployed with a deployment descriptor detailing the JMS topic the bean will use to listen for messages.

2 After the MDB is activated, a J2EE global transaction is started.

3 When a message arrives on the JMS Topic, it is then delivered to the MDB's onMessage() method.

4 The MDB calls the Hospital Controller's doMessage() method to process the message. This method first checks to see if this message is flagged for insertion into the Error Hospital.

- If so, it creates a set of new entries in the Error Hospital and returns success. There will be one new entry per RIB Message Node. Proceed to Step 6.

5 The Hospital Controller performs the following actions on each RIB Message Node found in the message.

- Checks the Error Hospital to see if there is an entry in it for this message, if so, this message is currently being retried.
- Checks the Error Hospital to see if there are entries in it for the same Message Family/Business Object ID combination.
  - If so, and this message is not being retried, this message is placed into the Error Hospital and a successful return is made. Proceed to Step 6.
- The Hospital Controller calls the MDB's handleMessage() method. This method invokes the RIB Binding subsystem to create an "injector" object. The Injector object is specific to the Message Family and Message Type. The RIB Binding subsystem first creates a RIB Payload object from the RIB Message Payload XML, which it then passes to the Injector through the inject() method. This method performs the required application specific logic to process the message. The inject() method returns the status of the message back to the MDB.

6 The MDB examines the status. If a failure has occurred, the transaction is marked rollback only. The message is marked as failed and control is returned to the MDB.

a On a failure,

- The MDB throws an exception to the MDB's container.
- A rollback of all database work is performed, and the message remains on the JMS Topic.
- The message is re-delivered to the MDB and steps 2, 3, and 4 are repeated.
- The MDB now recognizes that this is a re-delivery of a failure (retry message). It performs the actions detailed in Step 5 above and returns.

b On a successful return,

- The MDB checks the Error Hospital to see if this is a retry message. If so, it removes the message from the Error Hospital.

7 The MDB returns success to its container and the message is removed from the JMS Provider. A two-phase commit operation is performed with both the database(s) and JMS Provider committing all work. Steps 2-7 are repeated for each new message on the JMS Provider Topic.

RIBMessagePublisherEJB (Stateless Session Bean)

This EJB provides an interface into the RIB for converting a Payload into XML and publishing that message to JMS. The stubs and reference files needed to call this method are contained in the rib-client.jar provided by the RIB to the J2EE application.

The publish() method has the following signature:

```
public void publish(String family, String type, Payload payload,
ArrayList ids, ArrayList ris) throws PublishException{}
```

The message ***family*** and message ***type*** are passed in as Strings, along with the ***Payload*** object that contains the business data. The ***ids*** parameter is an ArrayList of Strings containing the business object ID, which is used for sequencing in the RIB. The ***ris*** parameter is an ArrayList of RoutingInfo objects, which are used for routing messages in the RIB.

Publishing Workflow

An overview of the publishing process is as follows:

1 The J2EE application determines that a message is to be published to the RIB. It creates a RIB Payload object to hold the business data. RIB Payload objects are message type specific and map directly to the RIB Message Payloads.

2 The J2EE application invokes the RIB Publishing EJB's publish() method. The RIB Publishing EJB is a stateless session bean. Parameters to this method include the RIB Message payload object, the Message Family, the Message Type, an array of Routing Infos, and an array of Business Object Ids.

3 The RIB Publishing EJB creates a new RIB Message from this information.

4 The RIB Error Hospital is checked for dependencies between the new message and the records in the Hospital. If dependencies are found, an exception is thrown, and the message is inserted into the Hospital.

5 The RIB Publishing EJB invokes the appropriate RIB Binding subsystem to create the XML Message Data for the RIB Message. The RIB Binding code also determines the correct JMS topic to use for publishing the message.

6 The RIB Publishing EJB publishes the RIB Message to a configured JMS Provider.

- If the publish fails, an exception is thrown. The RIB Publishing EJB tries to insert this message into the RIB Error Hospital, using the "JMS" REASON_CODE. If the insertion to the Hospital fails, an EJB Exception is thrown, and the transaction is rolled back. A PublishException is returned to the J2EE application, indicating that the publish was unsuccessful.
- If the publish succeeds, no PublishException is returned to the J2EE application.

7 The J2EE application determines if a PublishException was thrown from the RIB Publishing EJB.

- If so, an error appears in the application, and the database work is rolled back.
- If not, it completes its unit of work and a 2-phase commit operation is performed between any database(s) and the JMS server.

RIBMessageTafrEjb (MDB)

If messages have a requirement to be dropped, altered or routed before ultimate consumption, these types of MDBs will be deployed.

This MDB is responsible for listening to a JMS topic for messages, and upon finding a suitable message (based on the message selector), processing it through the onMessage() method. There is a different deployment of the MDB for every subscribing message family, as each MDB listens to a different topic on JMS. Also, in a multi-threaded environment, there could be a different deployment of the MDB for every thread for a specific message family.

The MDB passes the inbound message through the appropriate Java TAFR class. These TAFR classes have the ability to Transform, Filter and Route messages using the RIB's Java TAFR framework. If an Exception is returned from the TAFR class, the transaction will be rolled back to JMS.

TAFR Workflow

For the Tafr process, the process is as follows. It is very similar to the SeeBeyond TAFR e*Way process.

1 The Message Driven Bean (MDB) is deployed with a deployment descriptor detailing the JMS topic the bean will use to listen for messages and a Java TAFR class to use.

2 After the MDB is activated, a J2EE global transaction is started.

3 When a message arrives on the JMS Topic, it is then delivered to the MDB's onMessage() method.

4 The MDB calls Tafr Helper's convertMessage() method to process the message. This then passes the message through the following methods in this order. These methods have default implementations that do nothing. They should only be implemented if work needs to be done on the message.

- filterRibMessage() allows the Java Tafr class to drop unwanted messages. Messages can be dropped by evaluating any data in the RibMessage envelope or the message "payload" itself.
- transformRibMessage() allows the message data to be manipulated in any manner, such as translating one message (DTD) to another.
- routeRibMessage() allows the message to be routed to a particular JMS Topic based on evaluating some data in the RibMessage envelope or the message data itself.

5 If any failure occurs during this processing, a TafrException is thrown back to the TAFR MDB and the message is ultimately rolled back to JMS.

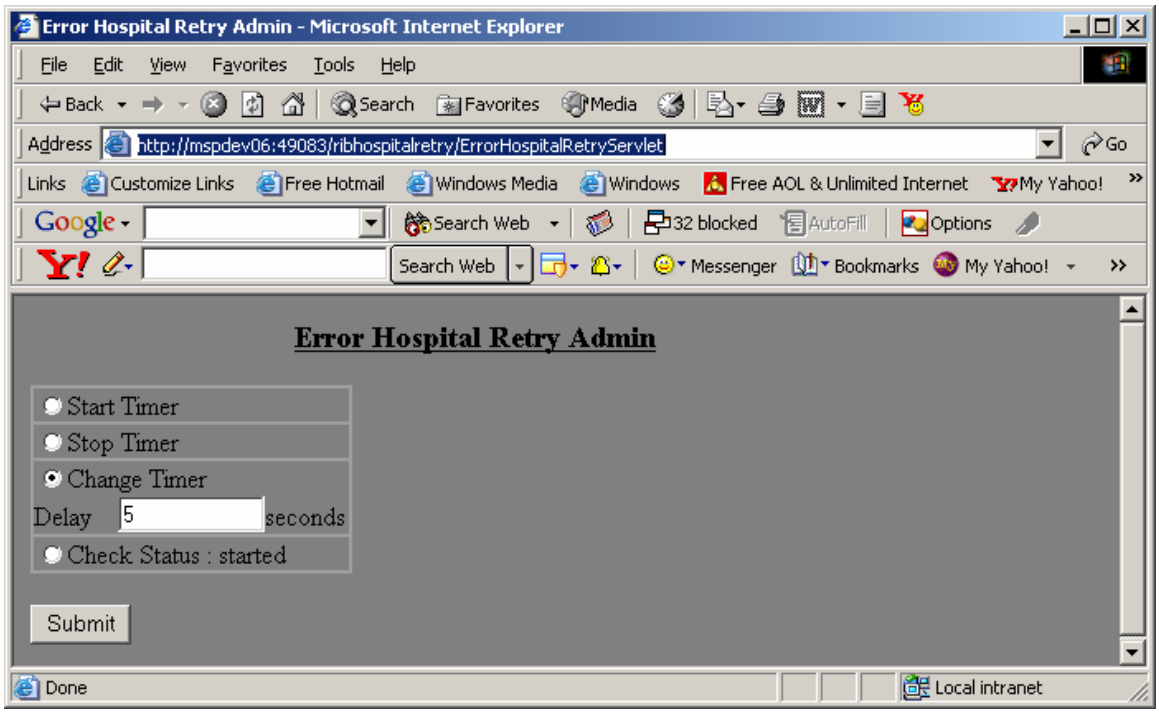
6 If successfully processed by the Java Tafr class, the RIB Tafr Framework will then publish this "transformed" message to the appropriate JMS Topic.

7 The MDB returns success to its container and the message is removed from the JMS Provider. A two-phase commit operation is performed with the JMS Provider committing all work. Steps 2-7 are repeated for each new message on the JMS Provider Topic.

ErrorHospitalRetryEjb (Stateless Session Bean)

The RIB guarantees that no published messages will be lost in the integration process. The RIB ErrorHospital is the mechanism that allows for message persistence. See Chapter 3 for more information on the RIB Error Hospital.

The ErrorHospitalRetryEjb is a stateless session enterprise bean that "retries" any messages that were put into the ErrorHospital. A timer triggers the ErrorHospitalRetryEjb to retry the messages. The timer can be configured and monitored by a servlet, ErrorHospitalRetryAdmin (http://<your host >:<port>/ribhospitalretry/ErrorHospitalRetryServlet). Initially when the rib EJB application starts, the timer also starts along with it. The initial timer interval duration can be configured by the property, "hospital.attempt.delay" in the rib.properties file. The timer can be started/stopped through the ErrorHospitalRetryAdmin. The status and interval duration time can also be changed/monitored through the ErrorHospitalRetryAdmin.



J2EE Application Overview

A J2EE Application that interfaces with the RIB provides an EJB interface in order to pass along application data through a Payload object (i.e. “inject” messages). This is required for a J2EE application to subscribe to messages from JMS.

InjectorEJB

The J2EE application defines an InjectorEJB based on the two remote classes provided by the RIB, InjectorEJBRemote and InjectorEJBRemoteHome. The RIB accesses the JNDI name for the application injector based on a property set in the rib.properties file.

(e.g. ‘app.jndi.injector=pkg.InjectorEJBName ‘)

The J2EE application receives a jar file (rib-client.jar) from the RIB for referencing shared objects, such as the remote InjectorEJB objects, the stubs and reference files for the RIBMessagePublisherEJB, and other classes used on the interface signatures. The J2EE application needs to have access to this jar in their classpath.

The RIB J2EE application also requires the stubs and other reference files created for the InjectorEJB API in order to call its methods. These classes should be contained in a jar file that the J2EE application produces and provides to the RIB.

The signature of the InjectorEJB should be as follows:

```
public void inject(String msgFamily, String msgType, Payload
payload)
throws InjectorException {}
```

In the inject() method, the InjectorEJB should find the appropriate injector class used to “inject” the Payload data into the database. These classes should be referenced using a properties file (injector.properties), and should be based on the message family and message type passed in. Each injector class should extend the RIB’s ApplicationMessageInjector interface provided in the rib-client.jar. This injector class will implement the inject() method from the ApplicationMessageInjector, and provide the business logic to “inject” the data to the database. If an exception occurs during this processing that requires the transaction to be rolled back, the InjectorEJB should throw an InjectorException with a detailed error description. This description will be shown in the RIB Error Hospital when the message has been rolled back.

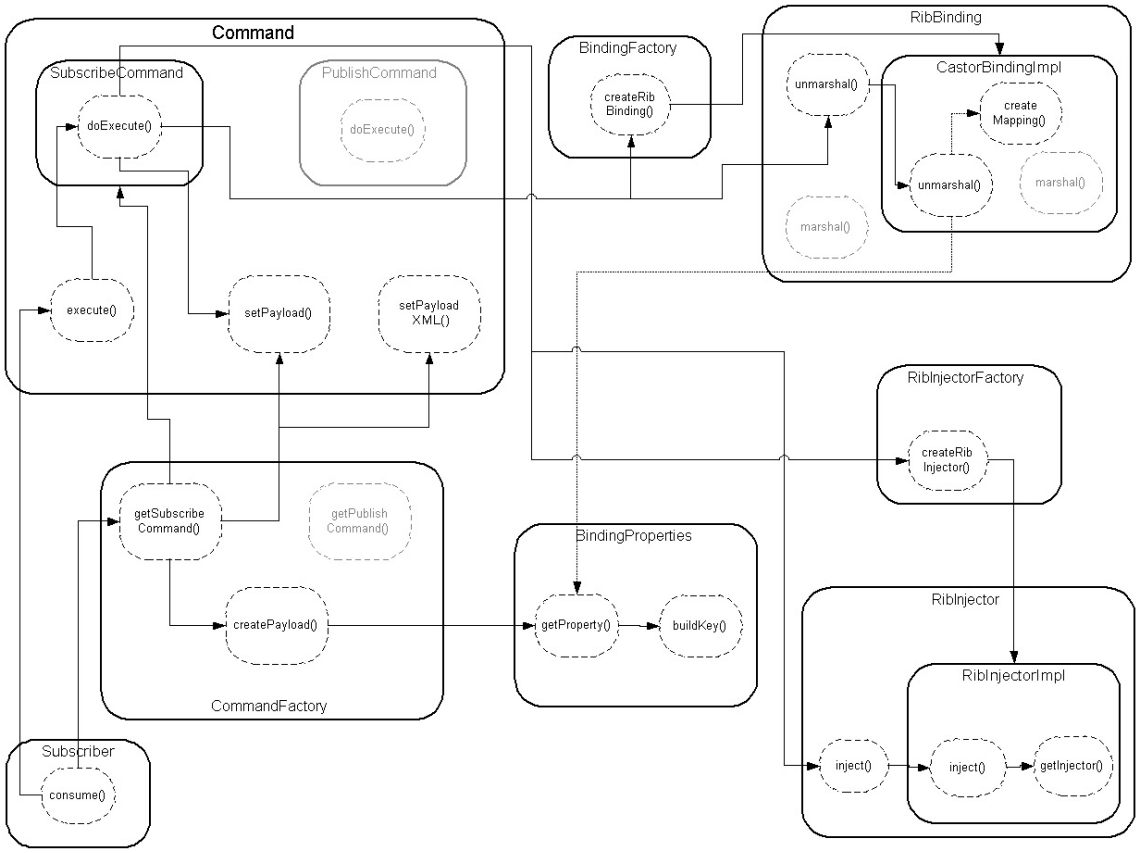
RIB Binding Overview

The RIB Binding code performs the necessary subscribing and publishing logic for XML Binding conversion between Java and XML, and for subscribing also calls the J2EE application injector logic.

The RIB Binding code contains commands that can be used for publishing and subscribing. The CommandFactory is called to either retrieve a SubscribeCommand or PublishCommand, and then the execute() method is called on the Command to perform the intended operations. The PublishCommand is used to marshal a Payload into an XML Message. The SubscribeCommand is used to unmarshal an XML message into a Payload, and inject that payload into the application code.

Properties files are used to determine the mapping between the message family and type, and the Java Payload object (and Castor Mapping file if used). See the section below on the properties files used by the Binding code for more information.

Subscriber Overview



XML is converted into a Java Payload object, and passed in to an application using the InjectorEJB's inject() method.

Subscriber Workflow

1 RIBMessageSubscriberEJB.handleMessage()

Calls the Subscriber.consume(***family*** String, ***type*** String, ***xml*** String, and the ***threadID*** String) method to consume the XML Message.

Subscriber.consume()

Calls the CommandFactory.getSubscribeCommand() to retrieve the Command object.

2 CommandFactory.getSubscribeCommand()

a A new SubscribeCommand object is created.

b CommandFactory.createPayload(): looks up the Java Payload class to use for the unmarshalling of the XML message. This value is retrieved using the BindingProperties class.

c SubscribeCommand.setPayload(): called with the instantiated Java Payload object.

d SubscribeCommand.setPayloadXML(): called with the message data XML String.

3 Subscriber.consume()

The Command object is returned back to Subscriber.consume(), where it calls the execute() method on the Command object. This method subsequently calls the SubscribeCommand.doExecute() method.

4 SubscribeCommand.doExecute()

The doExecute() method first looks up the implementation of RibBinding to use for unmarshalling the XML into a Java Object. The implementation is derived using a property in the rib.properties file. The unmarshal() command is called on the RibBinding implementation.

- RibBinding.unmarshal()

Unmarshals the XML using a pre-defined XML Binding tool.

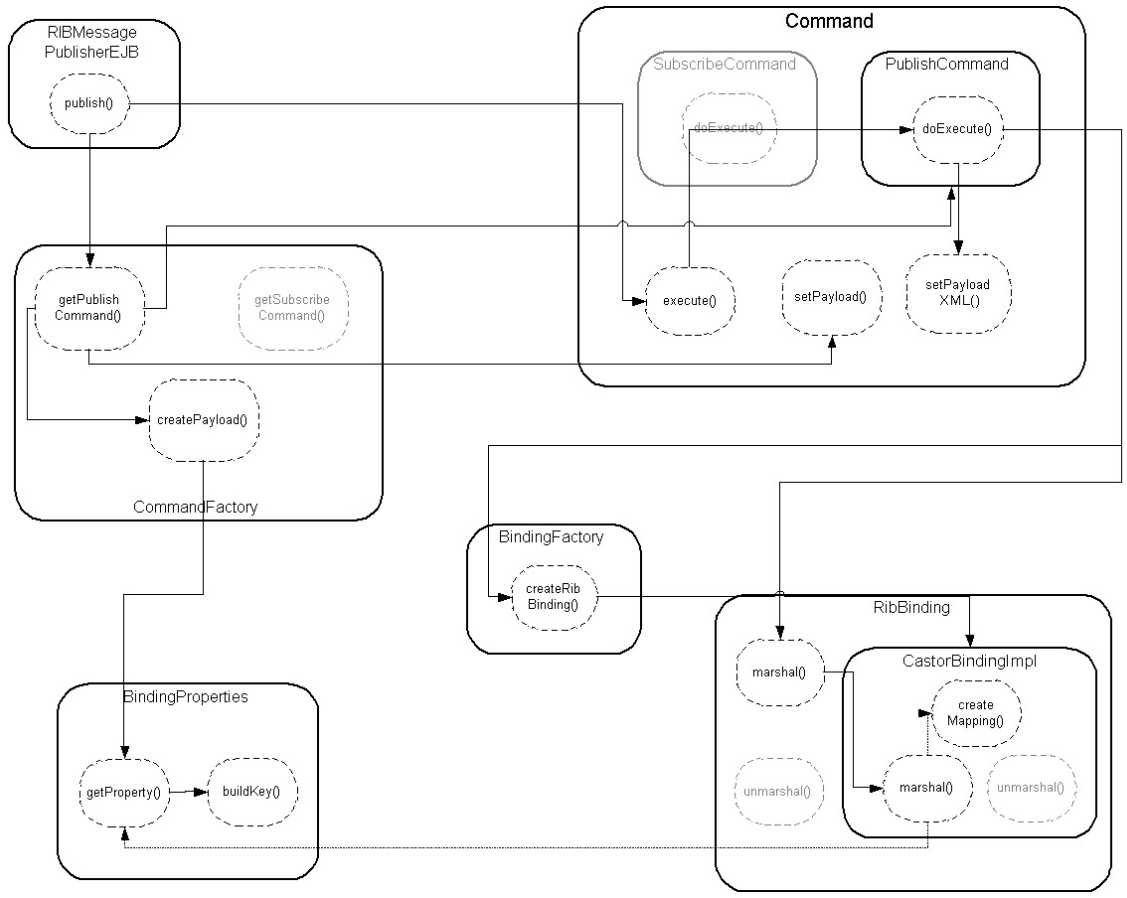
5 SubscribeCommand.doExecute()

The doExecute() method looks up which RibInjector class to use, using the RibInjectorFactory. This factory derives the correct implementation based on a setting in the rib.properties file. The inject() method is called on the RibInjector implementation.

- RibInjector.inject()

Calls inject(***family*** String, ***type*** String, and Payload ***payload***) on the application's injector class.

Publisher Overview



A Java Payload object is marshaled into an XML message. A RibMessages wrapper is created using the Payload XML message as the Message Data element. The XML message is published to JMS. On failure, the message is inserted into the RIB Error Hospital. If the message is not successfully inserted into the database, a PublishExeption is returned to the J2EE application.

Publishing Workflow

1 RIBMessagePublisherEJB.publish()

Calls the CommandFactory's getPublishCommand() method, which returns a Command object.

2 CommandFactory.getPublishCommand()

The value found in the payload.properties file is instantiated, and used in the setPayload(Payload ***payload***) method. This value is retrieved using the BindingProperties class.

3 RIBMessagePublisherEJB.publish()

Calls the Command.execute() method, which in turn calls the PublishCommand's doExecute() method.

4 PublishCommand

The doExecute() method marshals the Payload object into XML, and sets the setPayloadXML() method with the resulting XML String.

5 RIBMessagePublisherEJB.publish()

The RIBMessagePublisherEJB then creates the RibMessage XML. It uses the getPayloadXML() method to set the ***messageData***, along with the other elements such as message ***family***, message ***type***, ***ris*** (routing info), ***ids*** (business object ids), etc. This RibMessage is wrapped in a RibMessages element, and is published to JMS. If the publish to JMS fails, the message is inserted into the RIB Error Hospital. If for any reason the insertion into the database fails, the J2EE container rolls back the transaction, and sends a PublishException back to the J2EE application.

RIB Binding Classes

ApplicationMessageInjector

Interface used for the application's injector classes. Contains the inject() signature that must be implemented by each injector.

BindingFactory

The createRibBinding() method looks up the "`ribBindingImpl`" property from the rib.properties file that determines which implementation of the RibBinding interface to instantiate. This object is returned back to the calling method.

BindingProperties

This is a singleton class that looks up values in the payload.properties and binding.properties files. A value for a property is returned by calling the following static method:

```
BindingProperties.getInstance().getProperty(messageFamily,mess
ageType)
```

CastorBindingImpl

This class contains the unmarshal() and marshal() methods for the Castor XML Binding tool. This is the default implementation of the RibBinding interface.

This class also looks for a value in the binding.properties file, using the BindingProperties class. If a value is found for the message ***family*** and ***type***, the Castor Mapping file defined by the value is used in the unmarshal and marshal operations. If no value is found, the Castor Descriptor files are used. By default, no properties appear in the binding.properties file, as the Descriptor files are used.

Command

The Command class is the superclass for the SubscribeCommand and PublishCommand classes.

CommandFactory

The CommandFactory class creates either a SubscribeCommand or a PublishCommand, and populates the classes with the required values. It is responsible for using the BindingProperties to determine the Java Payload associated with a message ***family*** and ***type***, which the Command objects subsequently use in the unmarshal and marshal methods.

InjectorException

The InjectorException is used by the application InjectorEJB to return an exception to the RIB Binding code. This creates a rollback of the EJB transaction, and the message is sent to the RIB Error Hospital.

Payload

This is the common interface for all Java Payload objects. A Payload object used in RibBinding must extend this class, or the processing will fail.

PublishCommand

The PublishCommand holds the necessary information to call the RibBinding implementation's marshal() method, which transforms the Java Payload object to XML. This processing is performed inside of the doExecute() method.

PublishException

A PublishException is returned to the application upon failure of publishing a message to both JMS and the RIB Error Hospital. This will create a rollback of the EJB transaction.

RibBinding

This class is the interface for the RibBinding implementations. It allows for XML Binding tool independence, as the specific RibBinding implementation is the only place (besides the Java Payload objects) where Binding tool dependent code (such as code for Castor, JAXB, etc.) is referenced.

RibInjector

The RibInjector implementation allows for different implementations of the RibInjector class to be used for injecting a message into the application. This allows for J2EE and non-J2EE code to use the RibBinding code.

RibInjectorFactory

Determines which implementation of the RibInjector class to use, based on the value of the "`ribInjectorImpl`" property in the rib.properties file. It instantiates the class and returns the object.

RIBIntegrationException

Any exception occurring in the RIB Binding code is generally a RIBIntegrationException.

SubscribeCommand

The SubscribeCommand holds the necessary information to call the RibBinding implementation's unmarshal() method, which transforms the XML into a Java Payload object. It then determines the implementation of the RibInjector to use for application message injection. This processing is performed inside of the doExecute() method.

Subscriber

This class is called by the MDB for subscribing messages.

Properties Files

payload.properties

The payload.properties file maps the message family and message type Strings to a Java Payload class. This class is a Castor-generated Java object used for binding Java to XML. The key is the message family and the message type in uppercase characters, with a "." separator between the two. The equals sign, "=", is used to separate the key from the value. The value is the full class name (with package) of the Castor Java object. An example of this is shown below:

ASNOUT.ASNOUTCRE=com.retek.rib.binding.payload.ASNOutDesc

See the appendix for a sample payload.properties file.

binding.properties

The binding.properties file maps the message family and message type Strings to a Castor mapping XML file. This file can be used in place of the Castor Descriptor files that are generated in Castor for processing the XML binding between Java objects and XML. The Castor mapping file contains the rules for binding such as the ordering of elements or datatypes, etc. The key is built in the same way as the payload.properties file above, but the value will be the relative path to the Castor mapping file. This path can be seen in the jar file that contains the mapping files (retek-binding.jar). The relative path to the Castor mapping files is by default "com/retek/rib/binding/payload/" plus the filename. The file names are typically the same as the Java Payload object name, but with "Map" at the end of the name and an XML file format. An example of a binding.properties file entry is shown below:

ASNOUT.ASNOUTCRE=com/retek/rib/binding/payload/ASNOutDescMap.xml

rib.properties

The rib.properties file holds properties entries for configuring RIB code. This file is initially configured upon installation of the RIB application. See the RIB Installation Guide for more information on this file.

XML Binding Tool Independence

An XML Binding Tool other than Castor can be utilized in the RibBinding code. To implement a new XML Binding Tool:

1 If new Java Payload classes must be generated, they must extend the com.retek.rib.binding.payload.Payload class.

2 There must be a payload.properties file that defines the mapping between message family/type, and the Java Payload class. See the section above on the payload.properties file for more information on the file structure.

3 A new implementation of the RibBinding interface must be created. This class will contain at least two methods; a marshal() and an unmarshal(). Inside these methods, the XML Binding tool methods for marshal() and unmarshal() should be called.

The signatures of the RibBinding methods are shown below.

- The unmarshal method requires the message **family**, message **type**, Java **class name** of the Payload object, and the **xml** String to bind to the Java object.

```
public Payload unmarshal(
String msgFamily,
String msgType,
String className,
String xml) throws RIBIntegrationException {}
```

- The marshal method requires the message **family**, message **type**, and Java **Payload** object.

```
public String marshal(
String msgFamily,
String msgType,
Payload p) throws RIBIntegrationException {}
```

4 The rib.properties file must be edited to configure the RIB to use the new RibBinding implementation. Set the '`ribBindingImpl`' property to the new implementation class name. By default, this value is set to the CastorBindingImpl class.

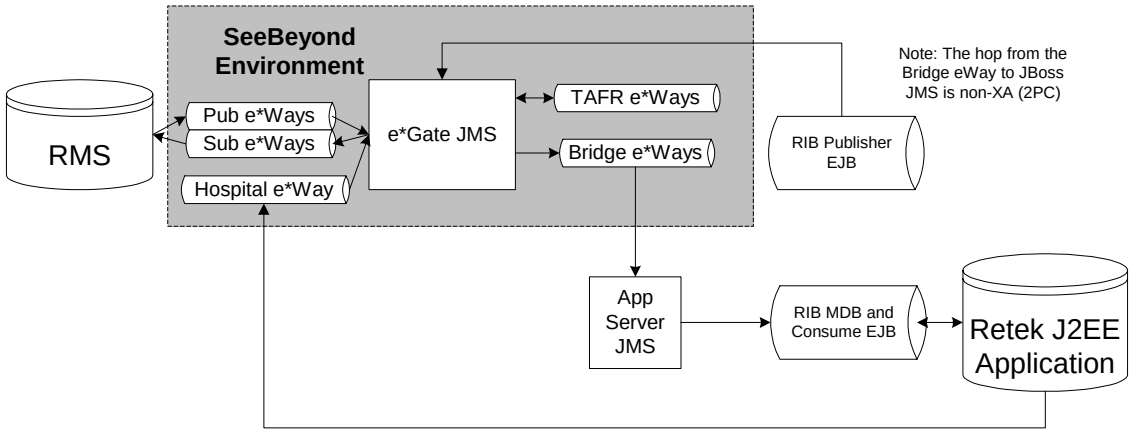
(e.g. ribBindingImpl=com.retek.rib.binding.CastorBindingImpl)

J2EE and SeeBeyond Bridging

In some cases, there needs to be application server specific bridges built between the SeeBeyond JMS and the Application Server supported JMS implementation.

The main problem for the J2EE platform is in a specific application server implementation of the JMS standard. Many application servers have their own "built-in" JMS implementations. In some cases, the application servers do not correctly use the JMS interface and hence not all application servers can use an externally provided JMS implementation. Many times the JMS implementation lacks an adequate XA compliant Two-Phase Commit interface with Message Driven Beans. In order to compensate for this, a set of Bridge e*Ways is needed to transfer the data from the SeeBeyond JMS to the application server specific JMS. A diagram of this configuration is seen below:

Bridging PL/SQL RIB - J2EE Solution



Note that this implementation also currently requires a SeeBeyond e*Way to poll the Error Hospital tables and retry failed messages and their dependent messages.

Appendix

Sample payload.properties file

ASNOUT.ASNOUTCRE=com.retek.rib.binding.payload.ASNOutDesc

BANNER.BANDLVSCDCRE=com.retek.rib.binding.payload.WSBanDlvScdDesc
BANNER.BANNERCRE=com.retek.rib.binding.payload.WSBannerDesc
BANNER.BANNERDEL=com.retek.rib.binding.payload.WSBannerRef
BANNER.BANNERMOD=com.retek.rib.binding.payload.WSBannerDesc
BANNER.CHANNELCRE=com.retek.rib.binding.payload.ChannelDesc
BANNER.CHANNELMOD=com.retek.rib.binding.payload.ChannelDesc

COBORES.CORESCANCRE=com.retek.rib.binding.payload.COResCanDesc
COBORES.CORESCRE=com.retek.rib.binding.payload.COResDesc

COCOGS.COGSCRE=com.retek.rib.binding.payload.WSCogsDesc

CODSRCPT.DSRCPTCRE=com.retek.rib.binding.payload.WSDSRcptDesc

CORETURN.CUSTRETSALECRE=com.retek.rib.binding.payload.CustRetSaleDesc

CORRESPONDENCE.CUSTCORRESCRE=com.retek.rib.binding.payload.CustCorresDesc

COSALE.CUSTSALECRE=com.retek.rib.binding.payload.CustSaleDesc

CUSTORDER.COCRE=com.retek.rib.binding.payload.CODesc
CUSTORDER.CODEL=com.retek.rib.binding.payload.CORef

CUSTRETURN.CORETCRE=com.retek.rib.binding.payload.CustRetDesc
CUSTRETURN.CORETDTLCRE=com.retek.rib.binding.payload.CustRetDesc
CUSTRETURN.CORETHDRCRE=com.retek.rib.binding.payload.CustRetDesc
CUSTRETURN.CALLTAGCRE=com.retek.rib.binding.payload.CallTagDesc

DIFFGRP.DIFFGRPDTLCRE=com.retek.rib.binding.payload.DiffGrpDtlDesc
DIFFGRP.DIFFGRPDTLMOD=com.retek.rib.binding.payload.DiffGrpDtlDesc
DIFFGRP.DIFFGRPHDRCRE=com.retek.rib.binding.payload.DiffGrpHdrDesc
DIFFGRP.DIFFGRPHDRMOD=com.retek.rib.binding.payload.DiffGrpHdrDesc

DIFFS.DIFFCRE=com.retek.rib.binding.payload.DiffDesc
DIFFS.DIFFMOD=com.retek.rib.binding.payload.DiffDesc

DSPO.DSPOMOD=com.retek.rib.binding.payload.DSPODesc
DSPO.DSPOSTATCRE=com.retek.rib.binding.payload.DSPOStatDesc

GIFTREG.GIFTREGACKCRE=com.retek.rib.binding.payload.GiftRegAckDesc
GIFTREG.GIFTREGUPDMOD=com.retek.rib.binding.payload.GiftRegUpdDesc

INVADJUST.INVADJUSTCRE=com.retek.rib.binding.payload.InvAdjustDesc

ITEMS.ISATTRCRE=com.retek.rib.binding.payload.WSISAttrDesc
ITEMS.ISATTRMOD=com.retek.rib.binding.payload.WSISAttrDesc
ITEMS.ISDLVBLKCRE=com.retek.rib.binding.payload.WSISDlvBlkDesc
ITEMS.ISDLVBLKMOD=com.retek.rib.binding.payload.WSISDlvBlkDesc
ITEMS.ISHIPRSDTLCRE=com.retek.rib.binding.payload.WSIShipRsDtlDesc
ITEMS.ISHIPRSHDRCRE=com.retek.rib.binding.payload.WSIShipRsHdrDesc
ITEMS.ISPERATTRCRE=com.retek.rib.binding.payload.WSISPerAttrDesc
ITEMS.ISPERATTRMOD=com.retek.rib.binding.payload.WSISPerAttrDesc
ITEMS.ISPERFNCLCRE=com.retek.rib.binding.payload.WSISPerFnClDesc
ITEMS.ISPERFNCLMOD=com.retek.rib.binding.payload.WSISPerFnClDesc
ITEMS.ISPERMXCHRCRE=com.retek.rib.binding.payload.WSISPerMxChrDesc
ITEMS.ISPERMXCHRMOD=com.retek.rib.binding.payload.WSISPerMxChrDesc
ITEMS.ITEMATTRCRE=com.retek.rib.binding.payload.WSItemAttrDesc
ITEMS.ITEMATTRDEL=com.retek.rib.binding.payload.WSItemAttrRef
ITEMS.ITEMATTRMOD=com.retek.rib.binding.payload.WSItemAttrDesc
ITEMS.ITEMBOMCRE=com.retek.rib.binding.payload.ItemBOMDesc
ITEMS.ITEMBOMMOD=com.retek.rib.binding.payload.ItemBOMDesc
ITEMS.ITEMCRE=com.retek.rib.binding.payload.ItemDesc
ITEMS.ITEMHDRMOD=com.retek.rib.binding.payload.ItemHdrDesc
ITEMS.ITEMLOCCRE=com.retek.rib.binding.payload.WSItemLocDesc
ITEMS.ITEMLOCMOD=com.retek.rib.binding.payload.WSItemLocDesc
ITEMS.ITEMLOCSCRE=com.retek.rib.binding.payload.WSItemLocsDesc
ITEMS.ITEMLOCSMOD=com.retek.rib.binding.payload.WSItemLocsDesc
ITEMS.ITEMSUPCRE=com.retek.rib.binding.payload.ItemSupDesc
ITEMS.ITEMSUPCTYCRE=com.retek.rib.binding.payload.ItemSupCtyDesc
ITEMS.ITEMSUPCTYMOD=com.retek.rib.binding.payload.ItemSupCtyDesc
ITEMS.ITEMSUPMOD=com.retek.rib.binding.payload.ItemSupDesc
ITEMS.ITEMUDAFFCRE=com.retek.rib.binding.payload.ItemUDAFFDesc
ITEMS.ITEMUDAFFMOD=com.retek.rib.binding.payload.ItemUDAFFDesc
ITEMS.ITEMUDALOVCRE=com.retek.rib.binding.payload.ItemUDALOVDesc
ITEMS.ITEMUDALOVMOD=com.retek.rib.binding.payload.ItemUDALOVDesc
ITEMS.ITMCARRSVCCRE=com.retek.rib.binding.payload.WSItmCarrSvcDesc
ITEMS.ITMCARRSVCMOD=com.retek.rib.binding.payload.WSItmCarrSvcDesc
ITEMS.ITMLOCATTRCRE=com.retek.rib.binding.payload.WSItmLocAttrDesc
ITEMS.ITMLOCATTRMOD=com.retek.rib.binding.payload.WSItmLocAttrDesc

MEDIA.DROPCODECRE=com.retek.rib.binding.payload.WSDropCodeDesc
MEDIA.DROPCODEDEL=com.retek.rib.binding.payload.WSDropCodeRef
MEDIA.MEDIACRE=com.retek.rib.binding.payload.WSMediaDesc
MEDIA.SOURCECODECRE=com.retek.rib.binding.payload.WSSourceCodeDesc
MEDIA.SOURCECODEDEL=com.retek.rib.binding.payload.WSSourceCodeRef

ORDER.ORDDATECRE=com.retek.rib.binding.payload.WSOrdDateDesc
ORDER.ORDDATEDEL=com.retek.rib.binding.payload.WSOrdDateRef
ORDER.ORDDATEMOD=com.retek.rib.binding.payload.WSOrdDateDesc

PAYMENTS.REFDPAYSTLMTCRE=com.retek.rib.binding.payload.RefdPayStlmtDesc

PENDRETURN.PENDRETCRE=com.retek.rib.binding.payload.PendRtrnDesc
PENDRETURN.PENDRETDTLCRE=com.retek.rib.binding.payload.PendRtrnDtlDesc

PENDRETURN.PENDRETDTLMOD=com.retek.rib.binding.payload.PendRtrnDtlDesc

SEEDDATA.CODEDTLCRE=com.retek.rib.binding.payload.CodeDtlDesc
SEEDDATA.CODEDTLMOD=com.retek.rib.binding.payload.CodeDtlDesc
SEEDDATA.CODEHDRCRE=com.retek.rib.binding.payload.CodeHdrDesc
SEEDDATA.CODEHDRMOD=com.retek.rib.binding.payload.CodeHdrDesc
SEEDDATA.DIFFTYPECRE=com.retek.rib.binding.payload.DiffTypeDesc
SEEDDATA.DIFFTYPEMOD=com.retek.rib.binding.payload.DiffTypeDesc

SOSTATUS.SOSTATUSCRE=com.retek.rib.binding.payload.SOStatusDesc

STORES.STORECRE=com.retek.rib.binding.payload.StoreDesc
STORES.STOREMOD=com.retek.rib.binding.payload.StoreDesc

UDAS.UDAHDRCRE=com.retek.rib.binding.payload.UDADesc
UDAS.UDAHDRMOD=com.retek.rib.binding.payload.UDADesc
UDAS.UDAVALCRE=com.retek.rib.binding.payload.UDAValDesc
UDAS.UDAVALMOD=com.retek.rib.binding.payload.UDAValDesc

VENAVL.VENCONIDSCRE=com.retek.rib.binding.payload.WSVenConIDsDesc
VENAVL.VENCONIDSMOD=com.retek.rib.binding.payload.WSVenConIDsDesc

VENDOR.VENCONTSCHCRE=com.retek.rib.binding.payload.WSSupContSchDesc
VENDOR.VENCONTSCHMOD=com.retek.rib.binding.payload.WSSupContSchDesc
VENDOR.VENDLVBLKCRE=com.retek.rib.binding.payload.WSSupDlvBlkDesc
VENDOR.VENDLVBLKMOD=com.retek.rib.binding.payload.WSSupDlvBlkDesc
VENDOR.VENDORADDRCRE=com.retek.rib.binding.payload.VendorAddrDesc
VENDOR.VENDORADDRMOD=com.retek.rib.binding.payload.VendorAddrDesc
VENDOR.VENDORCRE=com.retek.rib.binding.payload.VendorDesc
VENDOR.VENDORHDRMOD=com.retek.rib.binding.payload.VendorHdrDesc
VENDOR.VENDSATTRCRE=com.retek.rib.binding.payload.WSSupDsAttrDesc
VENDOR.VENDSATTRMOD=com.retek.rib.binding.payload.WSSupDsAttrDesc
VENDOR.VENPERFNCLCRE=com.retek.rib.binding.payload.WSSPerFnClDesc
VENDOR.VENPERFNCLMOD=com.retek.rib.binding.payload.WSSPerFnClDesc
VENDOR.VENPERRESCHACRE=com.retek.rib.binding.payload.WSSPerResChaDesc
VENDOR.VENPERTYPECRE=com.retek.rib.binding.payload.WSSupPerTypeDesc
VENDOR.VENPERTYPEMOD=com.retek.rib.binding.payload.WSSupPerTypeDesc

WH.WHATTRCRE=com.retek.rib.binding.payload.WSWHAttrDesc
WH.WHATTRMOD=com.retek.rib.binding.payload.WSWHAttrDesc
WH.WHCRE=com.retek.rib.binding.payload.WHDesc
WH.WHMOD=com.retek.rib.binding.payload.WHDesc

WOINT.WOINTCRE=com.retek.rib.binding.payload.WSWOIntDesc