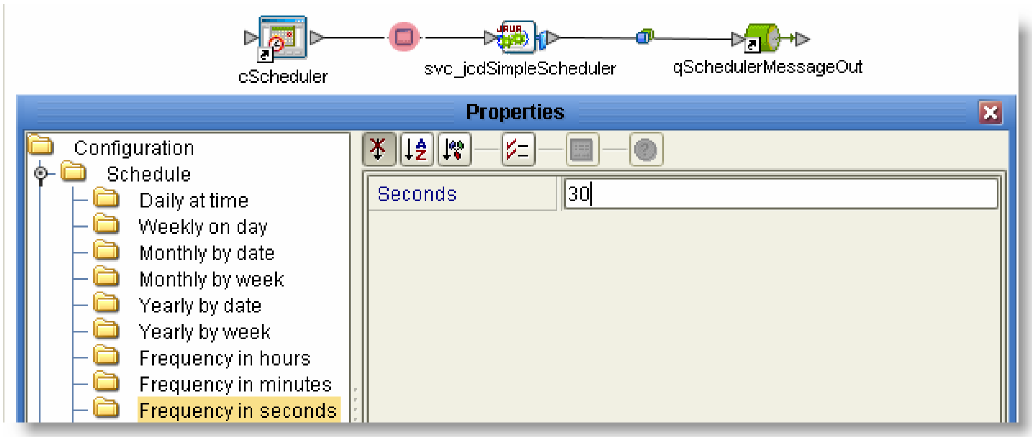


Configuration
- Schedule
  - Daily at time
  - Weekly on day
  - Monthly by date
  - Monthly by week
  - Yearly by date
  - Yearly by week
  - Frequency in hours
  - Frequency in minutes
  - Frequency in seconds

Seconds 30

Create a deployment profile, build and deploy the solution. Use the Enterprise Manager to inspect messages in qSchedulerMessageOut.

Note that the scheduler configuration is not particularly sophisticated and the content on the input message is the literal string “StaticString”.

@@@ talk briefly obout implementing a hearbeat with external scheduke for cron-like functionality @@@@@
@@@@@@@@@@@
[TBD]

9.4 Request/Reply

Some messaging solutions require that the recipient be active at the time of send, thereby guaranteeing to the sender that the message was received by the recipient. BEA MessageQ, a proprietary system from BEA Inc., is one such system. Other messaging solutions operate on a fire-and-forget, or a store-and-forward basis, expecting the messaging infrastructure to deliver each message to the intended recipient whether that recipient is active at the time of send or not. JMS is one such system. The major difference, from the architectural perspective, is the timing. In the former case a message is delivered ‘immediately’ or fails ‘immediately’ so the sender can branch as appropriate upon sending the message. In the latter case the message is delivered to the messaging system, which ‘immediately’ acknowledges that is has taken the responsibility for delivery to the ultimate recipient. That delivery, however, may take some time if the recipient is not active for some time or may not take place at all if the recipient never appears. The sender will never know what the ultimate outcome was.

In some situations a solution may be architected such that major pieces of functionality are built as functions, modules or services, accepting some input message, performing some processing and producing some result. It is necessary, to implement such components, that the invocation mechanism allows the invoker to invoke a component, provide the input data it needs, wait for the execution to complete and receive the result. This interaction is what [EIP] calls the Request/Reply pattern.

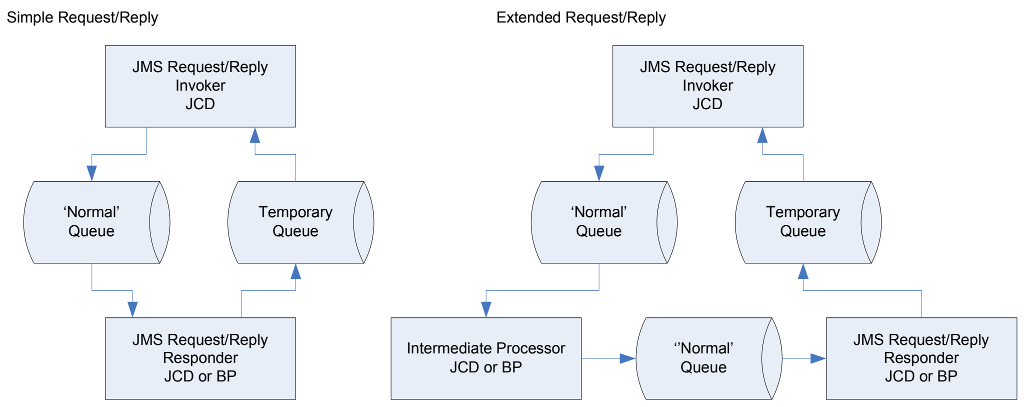
In general, one could implement a request/reply pattern using any end-points, with or without messaging infrastructure, as long as the requestor was engineered to make a request, wait for the reply and receive it. Even file exchange can support the request/reply pattern. In fact the Australian Energy Industry-developed HokeyPokey protocol uses the File Transfer Protocol (FTP) to implement a request/reply pattern for submitting XML documents. The sending component places a file containing an XML Document in a 'malibox' of an Energy Hub, then polls the 'outbox' at the Energy Hub for an Acknowledgment file. The exchange is not complete until the Acknowledgment file is received or a timeout occurs. This is a classic request/reply implementation.

Java CAPS provides a number of mechanisms to implement Request/Reply pattern. Which of the different mechanisms is appropriate will depend on the problem that needs to be solved. Whilst, as said previously, any end-point type can be used to implement the request/reply pattern, we will discuss only the more common/useful/interesting mechanisms – JMS Request/Reply, HTTP Request/Reply. SOAP Request/Reply, Web Service Invocation and TCP/IP Request Reply.

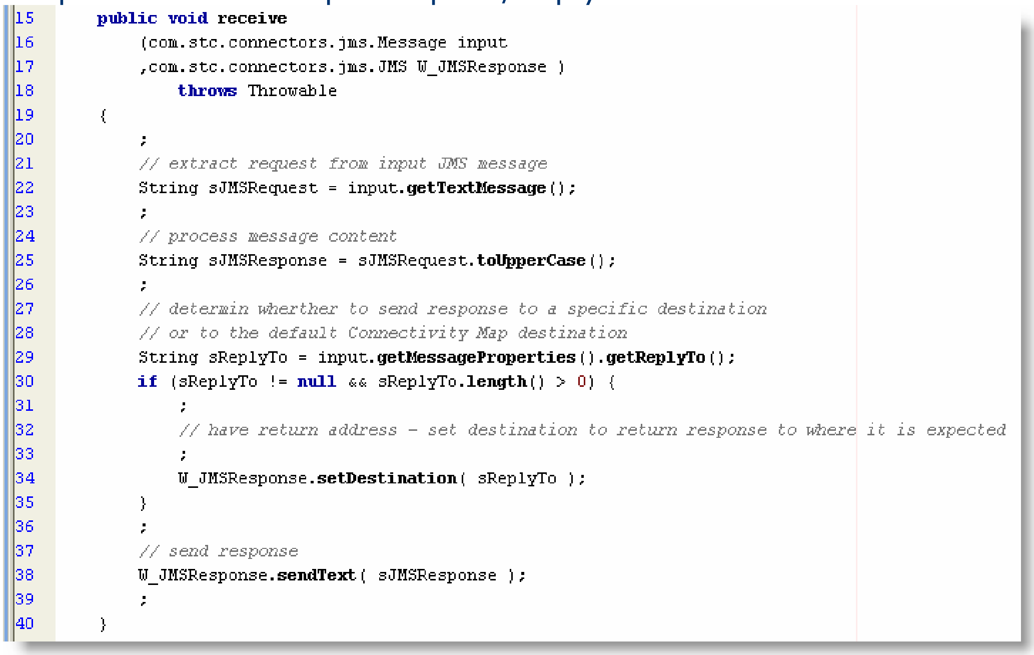
9.4.1 JMS Request/Reply

Whilst JMS is typically used to build store-and-forward messaging solutions, it also supports implementation of Request/Reply solutions using Temporary JMS Destinations. In Java CAPS a Java Collaboration can both be a requester and a responder in a request/reply configuration. An eInsight Business Process can only be a responder since no eInsight service exists that would allow an eInsight BP to invoke JMS Request/Reply functionality. All is not lost, however. Since an eInsight Business Process can invoke a 'New Web Service' Java Collaboration as an activity, and a Java Collaboration can invoke JMS Request/Reply functionality a JCD 'wrapper' can be used to overcome this limitation.

Picture the following models:


Simple Request/Reply
JMS Request/Reply Invoker JCD
'Normal' Queue
Temporary Queue
JMS Request/Reply Responder JCD or BP
Extended Request/Reply
JMS Request/Reply Invoker JCD
'Normal' Queue
Temporary Queue
Intermediate Processor JCD or BP
''Normal' Queue
JMS Request/Reply Responder JCD or BP

The receive method of a Java Collaboration that serves as a Responder in the simple Request/Reply model would look like this:

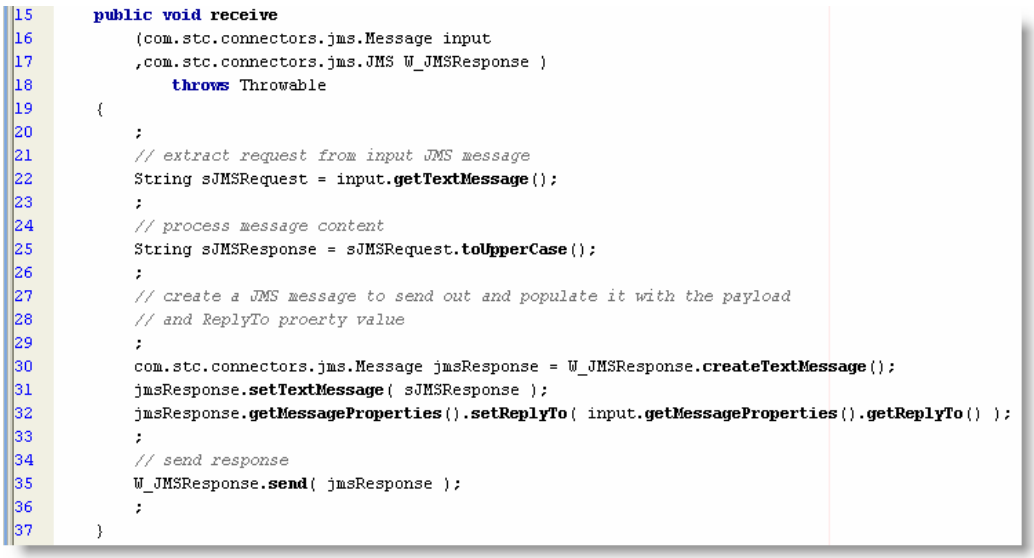

```
15      public void receive
16          (com.stc.connectors.jms.Message input
17          ,com.stc.connectors.jms.JMS W_JMSResponse )
18              throws Throwable
19      {
20          ;
21          // extract request from input JMS message
22          String sJMSRequest = input.getTextMessage();
23          ;
24          // process message content
25          String sJMSResponse = sJMSRequest.toUpperCase();
26          ;
27          // determin wherther to send response to a specific destination
28          // or to the default Connectivity Map destination
29          String sReplyTo = input.getMessageProperties().getReplyTo();
30          if (sReplyTo != null && sReplyTo.length() > 0) {
31              ;
32              // have return address - set destination to return response to where it is expected
33              ;
34              W_JMSResponse.setDestination( sReplyTo );
35          }
36          ;
37          // send response
38          W_JMSResponse.sendText( sJMSResponse );
39          ;
40      }
```

Needless to say a responder in a real solution would do something more interesting that converting the request string to upper case.

Note that this particular collaboration could be used as both a request/reply responder and as a 'regular' 'pick form one JMS Destination and deliver to another JMS Destination' collaboration.

By obtaining the value of the JMS ReplyTo property in the input message (line 29) and setting it as a destination for the response message (line 34) we are turning this collaboration into a request/response processor. If the component that submitted a message, which this collaboration is operating upon, did not set the ReplyTo property, the condition would be false and the response would go to the JMS Destination configured in the Connectivity Map.

Notice also that this collaboration does not need to be the one that directly interacts with the JMS Destinations set up by the requestor, as shown in the Extended Request/Reply model. There could be other collaborations and business processes operating on the message, with multiple JMS Destinations between the requestor and responder. As long as each component in the chain took care to propagate the value of the original ReplyTo property, set by the requestor, the response would still get delivered to the original requestor. Propagation of the ReplyTo, and other JMS Message properties requires a bit more work, as much as two extra lines:

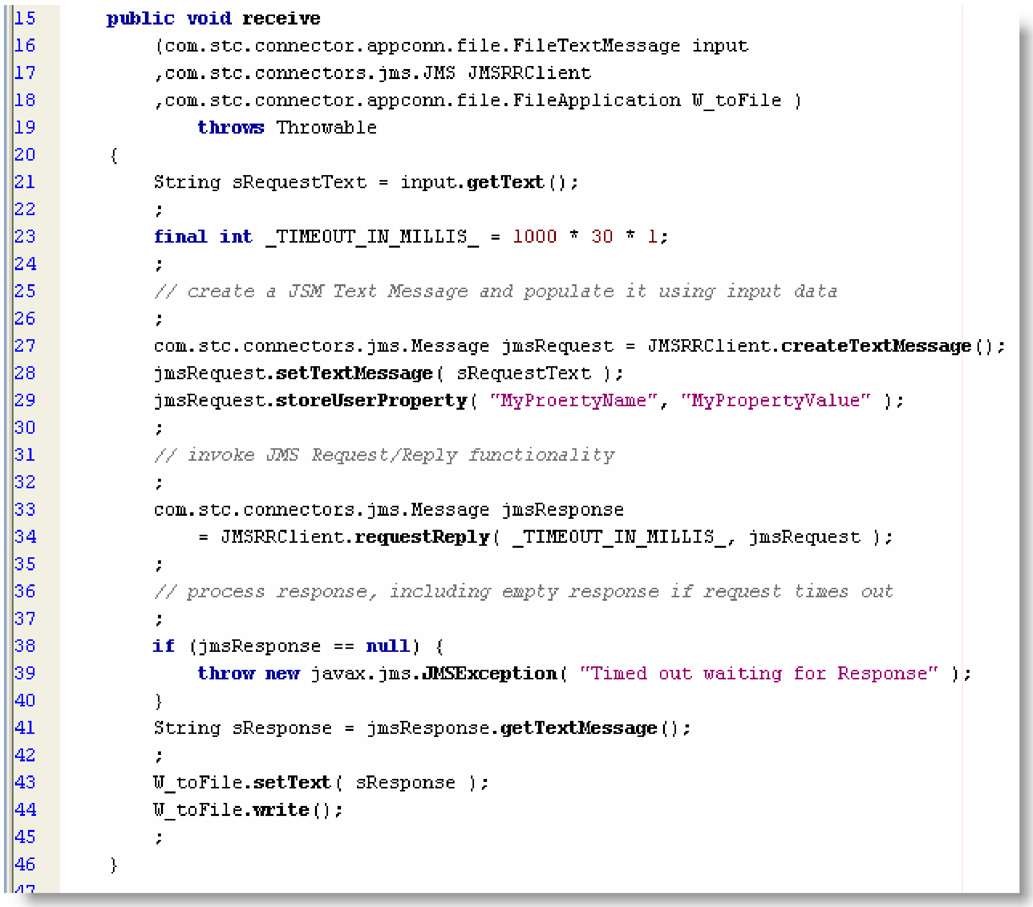


```
15      public void receive
16          (com.stc.connectors.jms.Message input
17          ,com.stc.connectors.jms.JMS W_JMSResponse )
18              throws Throwable
19      {
20          ;
21          // extract request from input JMS message
22          String sJMSRequest = input.getTextMessage();
23          ;
24          // process message content
25          String sJMSResponse = sJMSRequest.toUpperCase();
26          ;
27          // create a JMS message to send out and populate it with the payload
28          // and ReplyTo proerty value
29          ;
30          com.stc.connectors.jms.Message jmsResponse = W_JMSResponse.createTextMessage();
31          jmsResponse.setTextMessage( sJMSResponse );
32          jmsResponse.getMessageProperties().setReplyTo( input.getMessageProperties().getReplyTo() );
33          ;
34          // send response
35          W_JMSResponse.send( jmsResponse );
36          ;
37      }
```

In the previous example the sendText() method of the JMS Connector object was used to directly send the response string. In order to set properties a message object is required (line 40). The payload and the properties of the object are set (lines 41 and 42) and the message object is sent using the send() method. Needless to say other properties, including User-defined properties can be set before sending the message.

The JMS Request/Reply functionality relies on Java CAPS creating a Temporary JMS Destination, under the hood as it were, and transparently setting the ReplyTo property of the request message to the name of that destination. The requestReply() method puts the request message to a regular JMS Destination, named in the Connectivity Map and performs a blocking receive on the temporary destination. Once the message is received, or the time expires, the call returns to the collaboration.

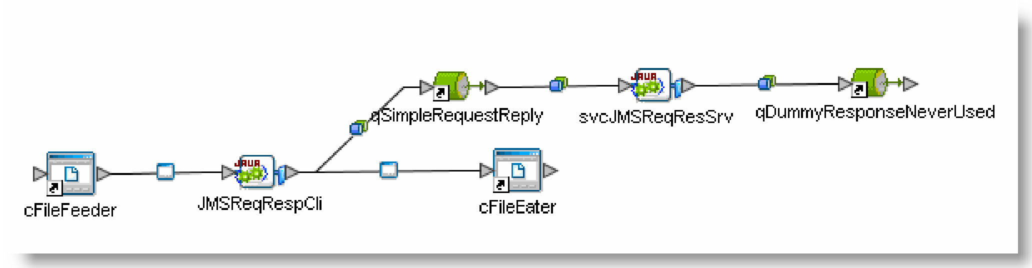
The receive method of a basic Java Collaboration that invokes a JMS Request/Reply functionality would look like this:



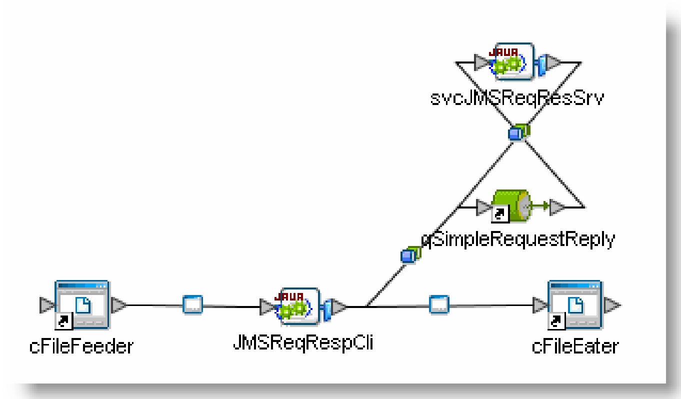
```
public void receive
    (com.stc.connector.appconn.file.FileTextMessage input
    ,com.stc.connectors.jms.JMS JMSRRClient
    ,com.stc.connector.appconn.file.FileApplication W_toFile )
        throws Throwable
{
    String sRequestText = input.getText();
    ;
    final int _TIMEOUT_IN_MILLIS_ = 1000 * 30 * 1;
    ;
    // create a JSM Text Message and populate it using input data
    ;
    com.stc.connectors.jms.Message jmsRequest = JMSRRClient.createTextMessage();
    jmsRequest.setTextMessage( sRequestText );
    jmsRequest.storeUserProperty( "MyProertyName", "MyPropertyValue" );
    ;
    // invoke JMS Request/Reply functionality
    ;
    com.stc.connectors.jms.Message jmsResponse
        = JMSRRClient.requestReply( _TIMEOUT_IN_MILLIS_, jmsRequest );
    ;
    // process response, including empty response if request times out
    ;
    if (jmsResponse == null) {
        throw new javax.jms.JMSException( "Timed out waiting for Response" );
    }
    String sResponse = jmsResponse.getTextMessage();
    ;
    W_toFile.setText( sResponse );
    W_toFile.write();
    ;
}
```

This collaboration is triggered by a File eWay and ultimately writes its output to a file using a File eWay. The content of the input file is set as the content of a request message. The JMS Request/Reply method is invoked with a timeout and a request message. The method will return a JMS message with the response or null if timeout occurred. The input message could be delivered by means other than a File eWay, and it could be pre-processed before being sent as a request. The response could be pos-processed and sent to some destination other than a File eWay. The point is that a JMS Request/Reply client is quite simple to implement in a Java Collaboration.

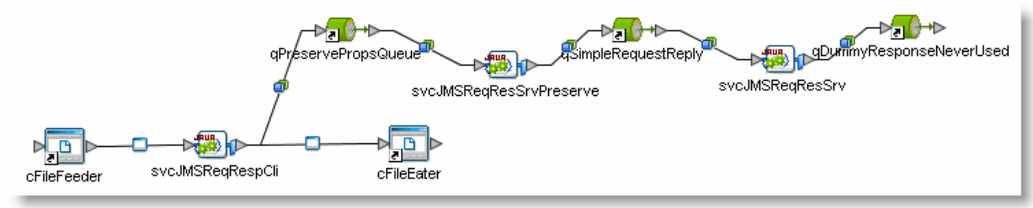
For the Simple Request/Reply model the connectivity Map will look like this:



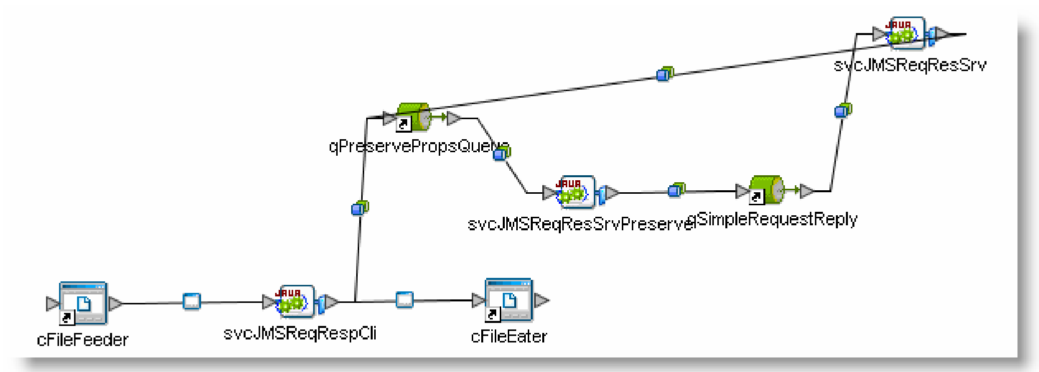
Note the final qDummyResponseNeverUsed. This JMS Destination is never used because the collaboration explicitly sends the response message to the JMS Destination whose name is specified in the JMS ReplyTo property – in this case it will be the name of the temporary destination created for the requester by the JMS Message Server. Note also that the Connectivity Map, which could otherwise be used as a good reflection of real connections, no longer accurately depicts the interactions that take place. The literal 'Dummy' is added to the name of the unused destination to give a strong hint that code may have to be inspected to discover what is actually happening. We could have, without loss of functionality, produced a Connectivity Map where the output of the service svcJMSReqResSrv would be connected to the qSimpleRequestReply. This could, perhaps be better for the simple case as it would suggest a request/response relationship.



For Extended Request/Reply mode the Connectivity Map will look like this:

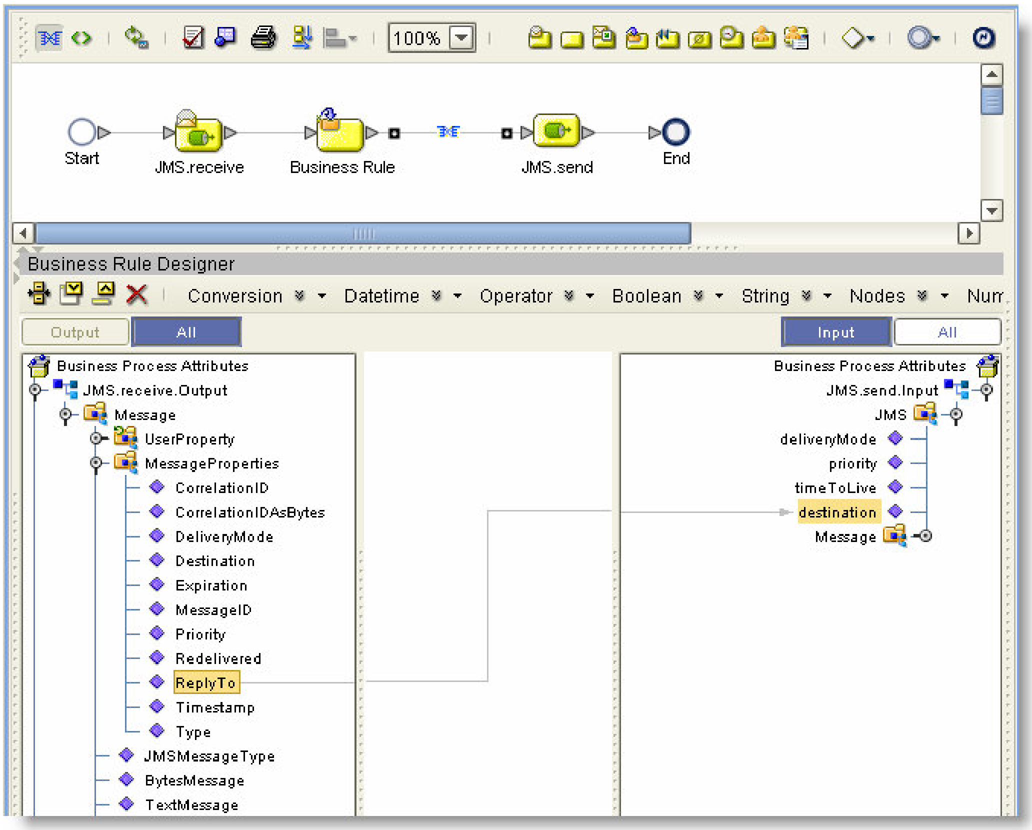


Here too we could have connected the output of svcJMS ReqResSvc service to qPreservePropsQueue, to suggest that request/response pattern is used.

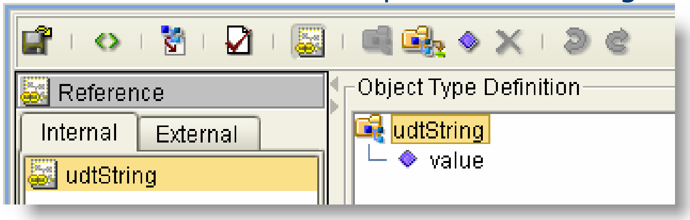


An important property of the temporary queue, created by the requestReply() method, is that it exists only as long as its creator exists. In this case, if the collaboration that invokes the requestReply() method exits, because the requestReply() timed out, for example, the temporary queue will be destroyed. If the responding service attempts to put a response message to the response queue it will receive an exception because the temporary response queue no longer exists. This is a very desirable characteristic in request/response scenarios where requestor will not wait longer than a certain amount of time. If appropriately designed, the responder will discard late responses on exception. Note, however, that the responses may be lost by design. Not also that should the system fail, whilst request processing is in progress, message loss will occur.

Much as a Java Collaboration can be used to implement the responder logic so too an eInsight Business Process can be used for this purpose. Similarly, and even more simply than in the case of a Java Collaboration, the ReplyTo property needs be copied from the input JMS Message to the Destination property of the output JMS Message. The infrastructure will take care of the rest.

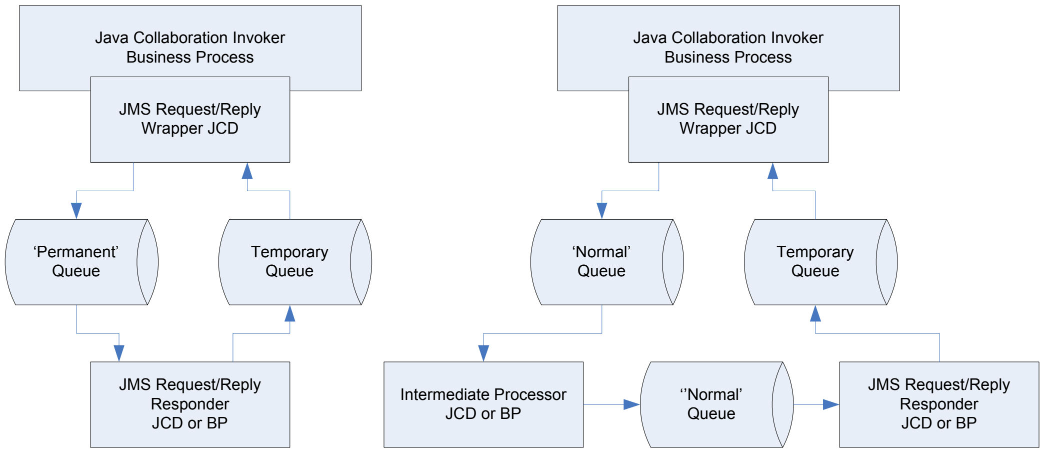


Where building a JMS Request/Reply responder in eInsight is easy, building a JMS Request/Reply requestor is perhaps harder than it needs to be. As mentioned before, it is necessary to write a Java Collaboration to wrap the call to the JMS requestResponse() method and invoke that collaboration as an activity in the business process. The collaboration will look almost exactly like the one presented earlier except it will be designed as a "New Web Service" collaboration, and the input and output will be message structures rather than connectors. How to create an OTD to use as the input and output will not be covered here. Let's just assume we have a user-defined OTD that represents a string.

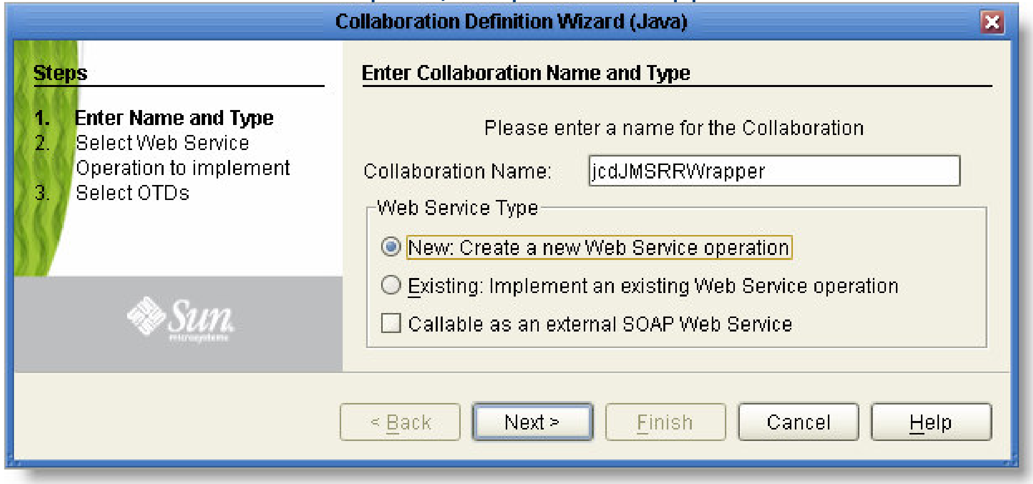


Here is a schematic of a Business Process-based JMS Request/Response solution discussed in the following section:

Message Exchange Patterns


Java Collaboration Invoker Business Process
JMS Request/Reply Wrapper JCD
'Permanent' Queue
Temporary Queue
JMS Request/Reply Responder JCD or BP
Java Collaboration Invoker Business Process
JMS Request/Reply Wrapper JCD
'Normal' Queue
Temporary Queue
Intermediate Processor JCD or BP
''Normal' Queue
JMS Request/Reply Responder JCD or BP

Let's create the JMS Request/Response Wrapper Collaboration:


Collaboration Definition Wizard (Java)
Steps
1. Enter Name and Type
2. Select Web Service Operation to implement
3. Select OTDs
< Back
Next >
Finish
Cancel
Help

Enter Collaboration Name and Type

Please enter a name for the Collaboration

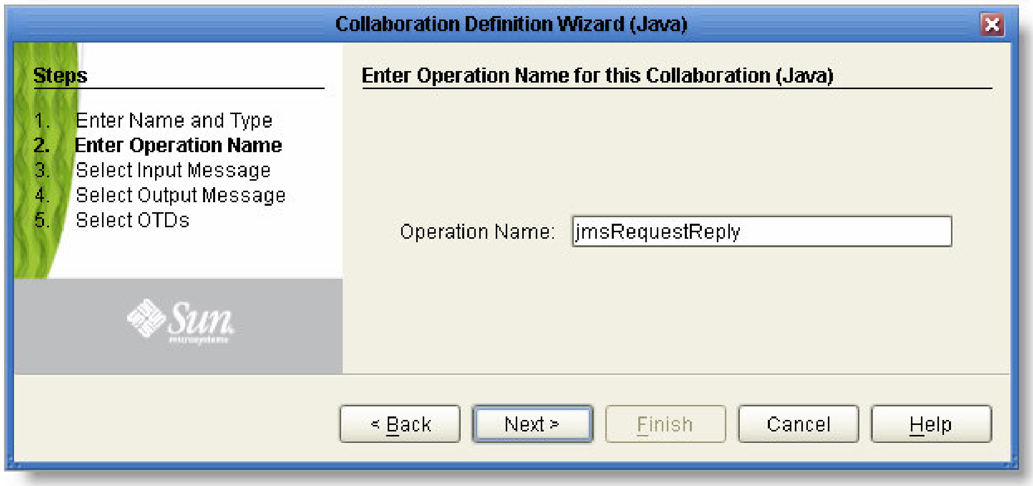
Collaboration Name: jcdJMSRRWrapper

Web Service Type

[● New: Create a new Web Service operation]

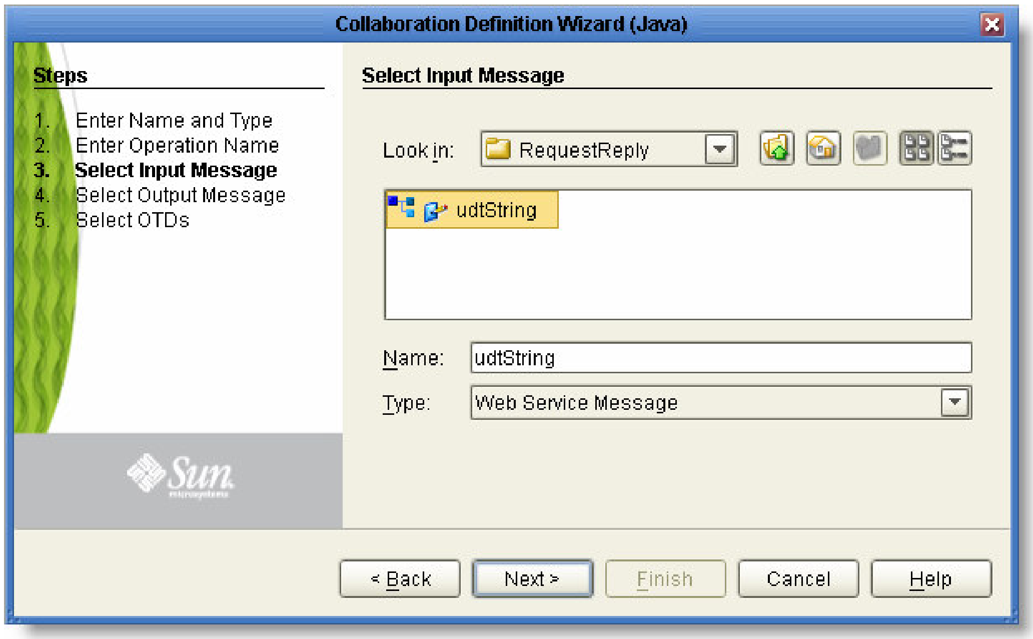
○ Existing: Implement an existing Web Service operation

☐ Callable as an external SOAP Web Service



Enter Operation Name for this Collaboration (Java)

Operation Name: jmsRequestReply



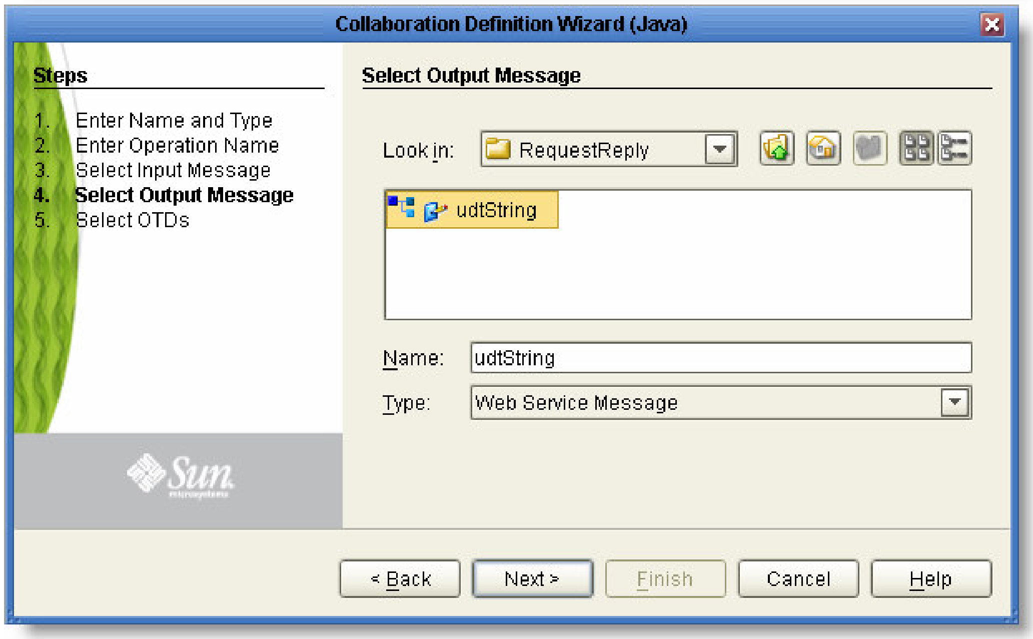
Select Input Message

Look in: RequestReply

udtString

Name: udtString

Type: Web Service Message



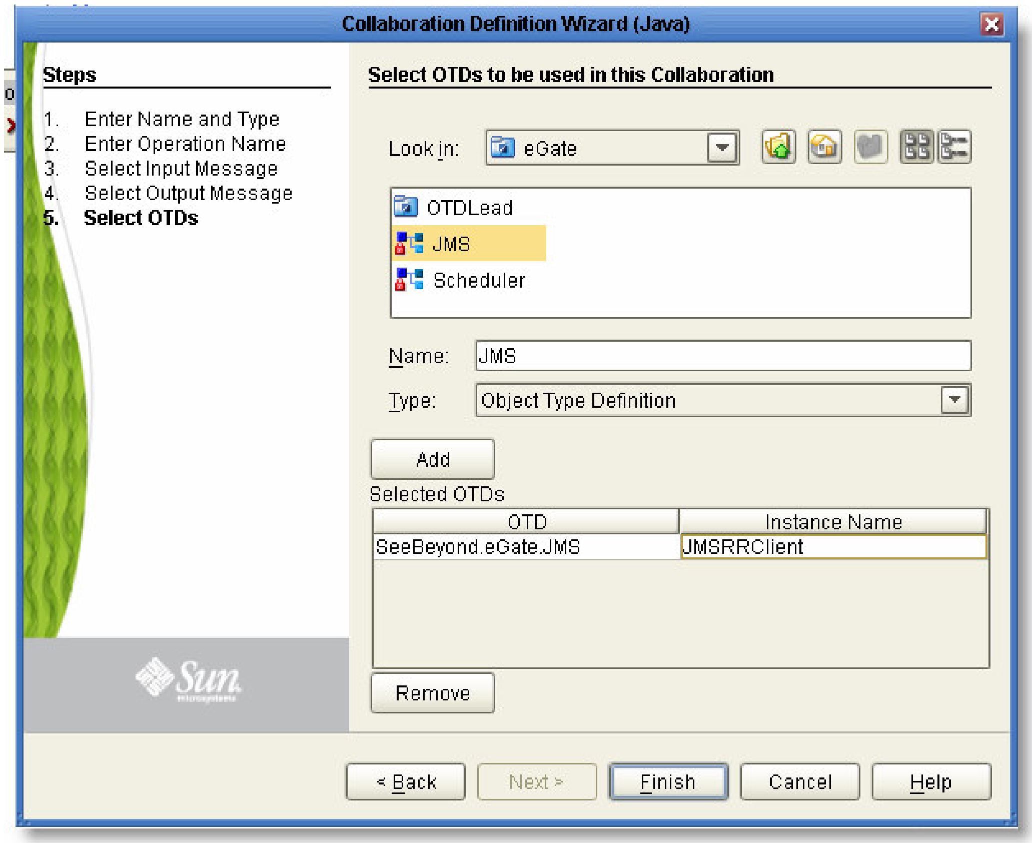
Select Output Message

Look in: RequestReply

udtString

Name: udtString

Type: Web Service Message

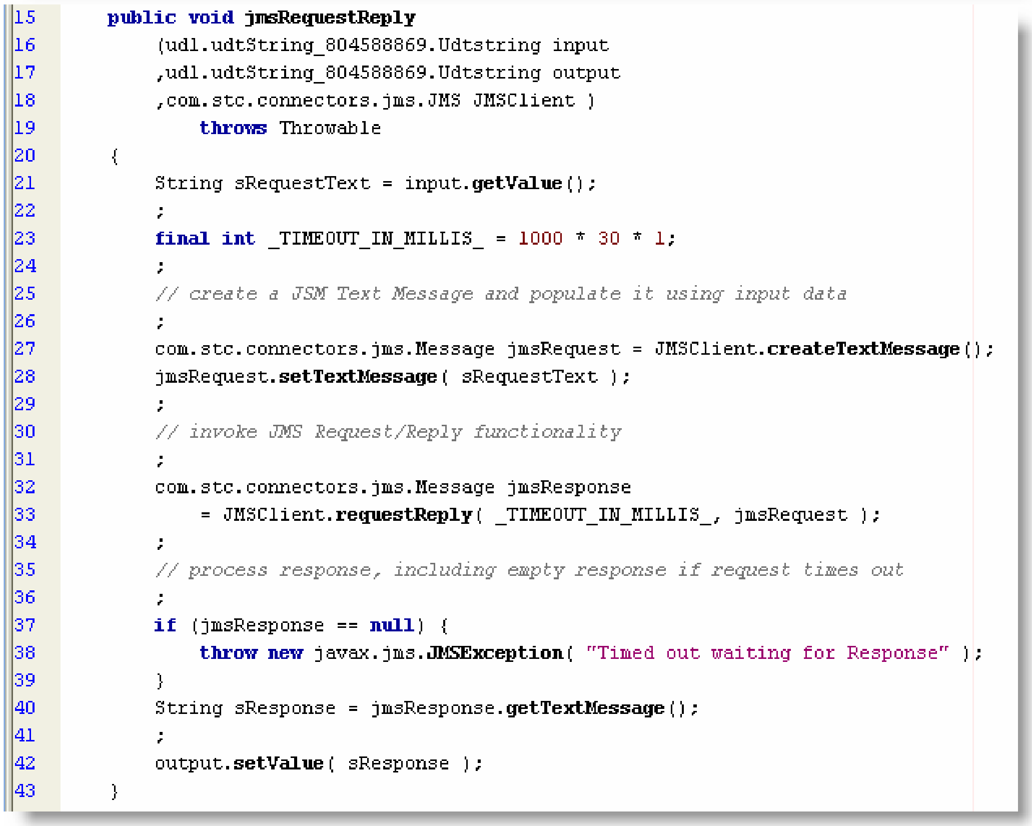


Name: JMS

Type: Object Type Definition

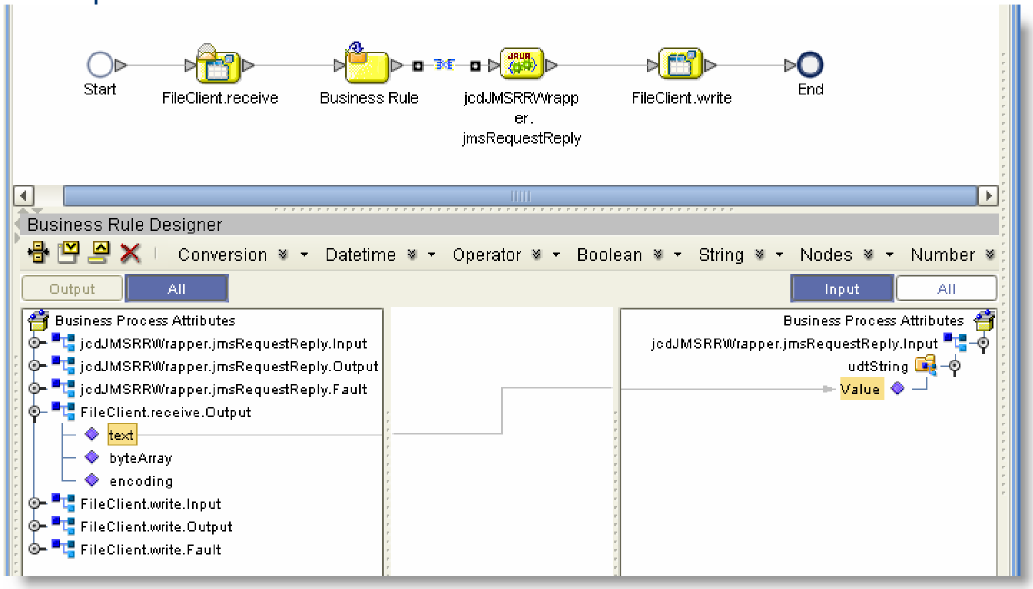
Selected OTDs

OTD	Instance Name
SeeBeyond.eGate.JMS	JMSRRClient

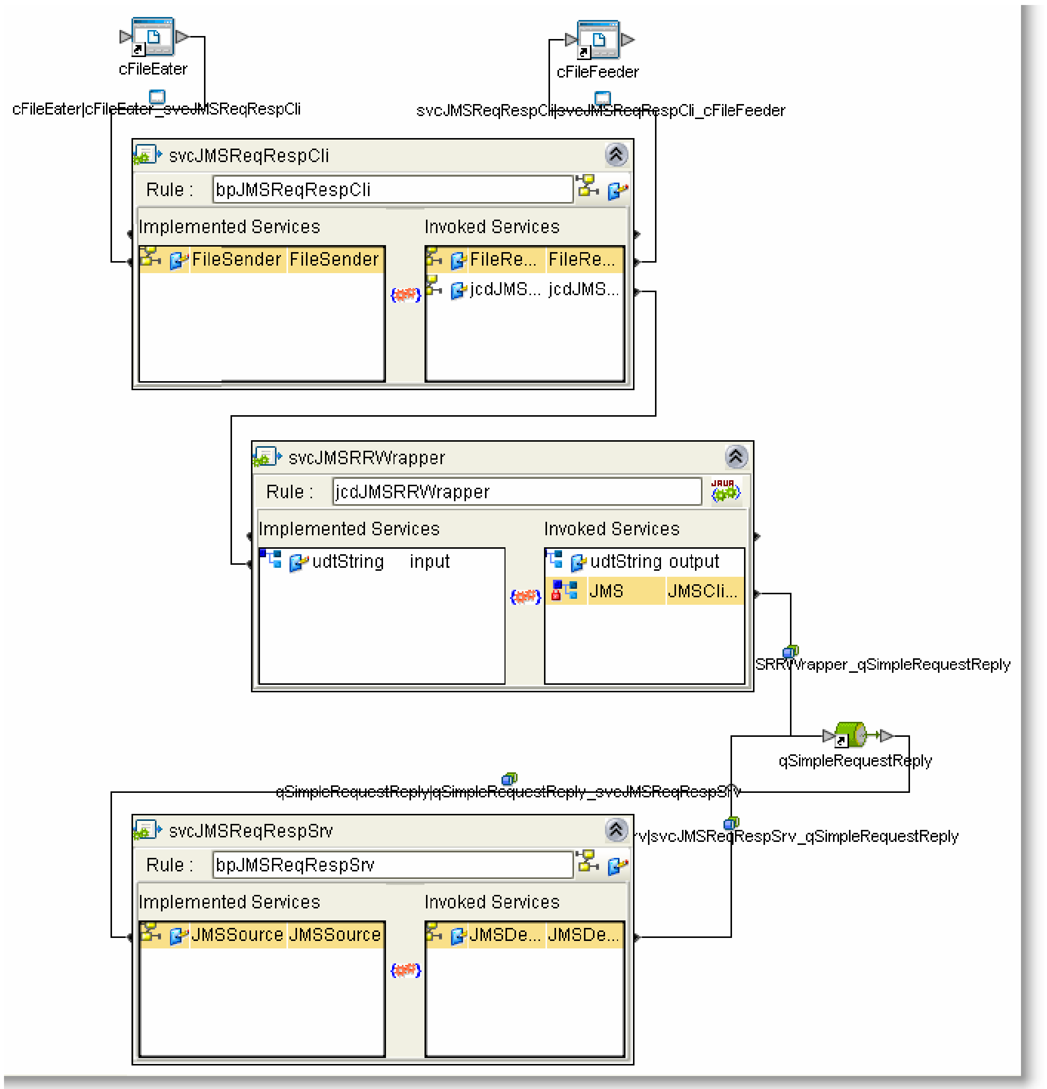


```
    public void jmsRequestReply
        (udl.udtString_804588869.Udtstring input
        ,udl.udtString_804588869.Udtstring output
        ,com.stc.connectors.jms.JMS JMSClient )
            throws Throwable
    {
        String sRequestText = input.getValue();
        ;
        final int _TIMEOUT_IN_MILLIS_ = 1000 * 30 * 1;
        ;
        // create a JSM Text Message and populate it using input data
        ;
        com.stc.connectors.jms.Message jmsRequest = JMSClient.createTextMessage();
        jmsRequest.setTextMessage( sRequestText );
        ;
        // invoke JMS Request/Reply functionality
        ;
        com.stc.connectors.jms.Message jmsResponse
            = JMSClient.requestReply( _TIMEOUT_IN_MILLIS_, jmsRequest );
        ;
        // process response, including empty response if request times out
        ;
        if (jmsResponse == null) {
            throw new javax.jms.JMSException( "Timed out waiting for Response" );
        }
        String sResponse = jmsResponse.getTextMessage();
        ;
        output.setValue( sResponse );
    }
```

Now drag the JCD operation onto the Business Process Editor canvas, assign input, use output and complete the process as might be required.


Start
FileClient.receive
Business Rule
jcdJMSRRWrapper.jmsRequestReply
FileClient.write
End
Business Rule Designer
Conversion
Datetime
Operator
Boolean
String
Nodes
Number
Output
All
Input
All
Business Process Attributes
jcdJMSRRWrapper.jmsRequestReply.Input
jcdJMSRRWrapper.jmsRequestReply.Output
jcdJMSRRWrapper.jmsRequestReply.Fault
FileClient.receive.Output
text
byteArray
encoding
FileClient.write.Input
FileClient.write.Output
FileClient.write.Fault
Business Process Attributes
jcdJMSRRWrapper.jmsRequestReply.Input
udtString
Value

The Connectivity Map is shown below. This Connectivity Map includes all components involved in the example. Since the client components and the server are sharing a JMS Destination the client components could be deployed using separate Connectivity Map and Deployment Profile from the server components.

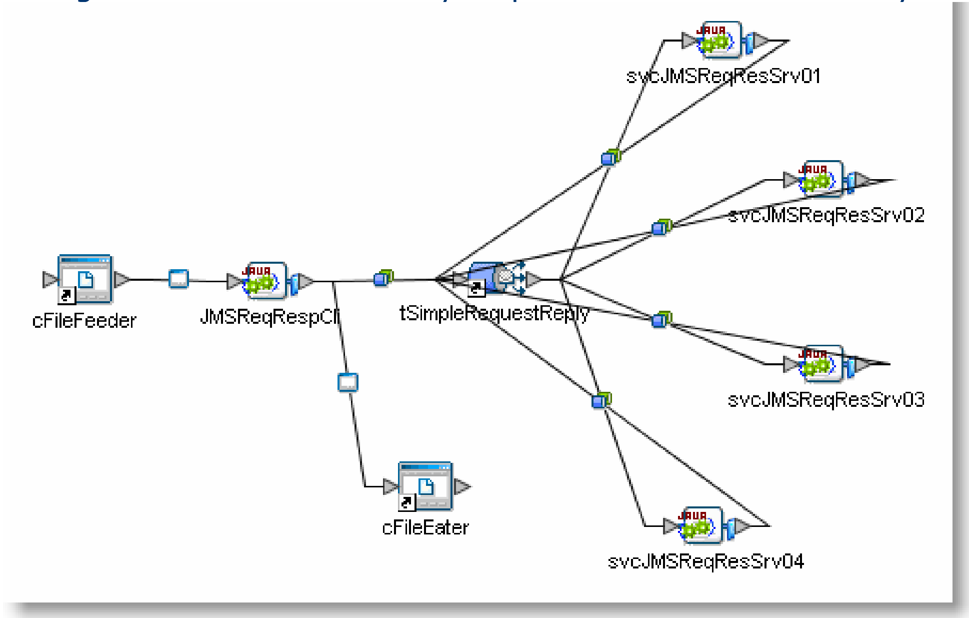


Ultimately, whether the responder is implemented as a Java Collaboration or a Business Process with a Java wrapper, the JMS-based Request/Response pattern can be used to construct reusable components.

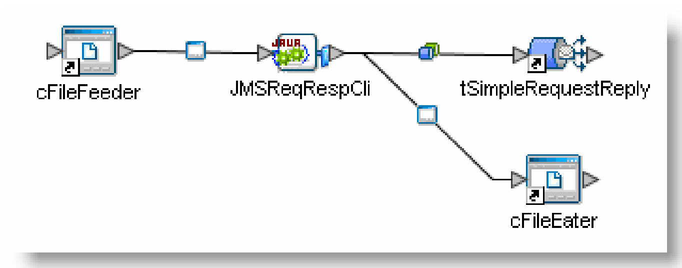
The requestReply() method uses a temporary JMS Destination, whose name is transparently set in the request message and is later used by the responder as the destination for responses. As mentioned, the temporary destination, and the messages within it, will be destroyed when the collaboration that created it exits. Since the name of the temporary JMS Destination is transparently generated and set in the ReplyTo property of the outgoing message it will a) be different from invocation to invocation and b) only the recipient of the message will have access to it. Note, also, that the requestReply() method variant, used in sample code, specifies a timeout parameter. The value of that parameter must be large enough to guarantee delivery of responses in normal circumstances.

The discussion above uses a somewhat awkward term JMS Destination to name what one would naturally call a Queue. This is because in the Java Collaboration code samples and Business Process screenshots there is no distinction between Queues and Topics, both of which are JMS Destinations. It is not until the Connectivity Map is being constructed that the actual JMS Destination type is specified. This is handy as the code is generic and destination type independent and because the same code, if general enough, can be used in solutions using Queues and ones using Topics.

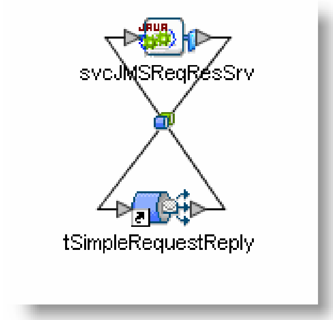
A form of an Auction Pattern, where the fastest responder wins, would be an interesting application of a JMS Request/Reply pattern, which uses Topics rather than Queues, and has multiple responders configured in the Connectivity Map. Here is the Connectivity Map:



JMS will deliver a copy of each message to all subscribers. Once a subscriber/responder completes the request and submits the response the requestor will complete, causing the temporary Topic, and any responses from other respondents, to be destroyed. The multiple responders could be deployed to different Integration Servers, likely on different physical machines, thus distributing workload so as to get the best response time possible. The Connectivity Map explicitly specifies 4 responders. To make this solution more flexible one would break up the Connectivity Map into two, a Requestor and a Responder, and create as many Responder Deployment Profiles as might be desired. Responders could then be added and removed by deploying and un-deploying Responder deployments.
Requestor Connectivity Map:



Responder Connectivity Map:



This implementation is very wasteful of resources because all responders eventually perform all the work required to process all messages, only to have all but one response discarded. If fastest possible response or fault tolerance is needed, however, this kind of implementation may be appropriate.

An important point to note about the JMS requestReply() method is that, at leastthrough version 5.1.1, both the requestor and the responder JMS Clients must be deployed to the same Message Server. If the responder must be deployed to a different JVM then other request/reply mechanisms must be considered instead.

9.4.2 HTTP Request/Reply

The Hypertext Transfer Protocol (HTTP) is the embodiment of the Request/Reply pattern. A HTTP GET or a HTTP POST request is submitted to a HTTP Server, which returns a response. The expression HTTP Server is used deliberately to describe a server that implements the HTTP Protocol. To use the expression Web Server would be to invite confusion. Event in the early days of HTTP 0.9 a HTTP Server could return content other than text/html, which is what a 'web page', as implied in the expression 'web server', would be. With support for the Multipurpose Internet Mail Extensions (MIME) content-type specification and handling, the range of content types that can be returned by a HTTP Server is virtually limitless. These properties of HTTP are taken advantage of to implement Request/Response solutions that deal with content other than the Hypertext Markup Language [HTML401]. HTTP makes provisions for PUT, DELETE, TRACE, CONNECT and OPTIONS