developerWorks®

Programming patterns for WebSphere Message Broker V6 timer nodes

Wayne Schutz

March 29, 2006

This article gives an overview of the WebSphere Message Broker V6 timer nodes and describes several programming patterns for using them. The article supplements the WebSphere Message Broker information center documentation on using the TimeoutNotification and TimeoutControl nodes.

Introduction

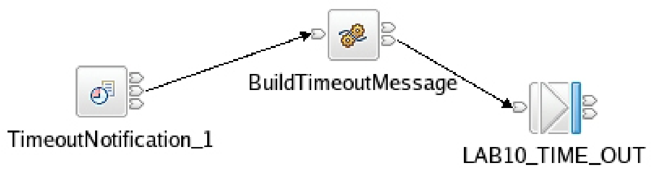
IBM® WebSphere® Message Broker V6 (hereafter called Message Broker) introduces two new built-in nodes: `TimeoutControl` and `TimeoutNotification`. These nodes enable you to initiate flows in Message Broker based on time-related events, such as using the `TimeoutNotification` node to run a message flow once an hour. There are no restrictions on the nodes that can be wired into a message flow that is initiated by a timer node.

Here is a brief overview of the two new nodes. Detailed technical information appears later in the article.

TimeoutNotification node

The `TimeoutNotification` node would be the first node in a timer-controlled message flow. It has no `In` terminal, which is normally where we would place an `MQInput, MqeInput, JMSInput` or similar node in a message flow. A property of this node is its unique identifier -- a 1 to 12 character label. This identifier must be globally unique within the broker instance that this node is deployed. `TimeoutNotification` nodes may be standalone or coupled with one or more `TimeoutControl` nodes.

Figure 1. TimeoutNotification node in a message flow



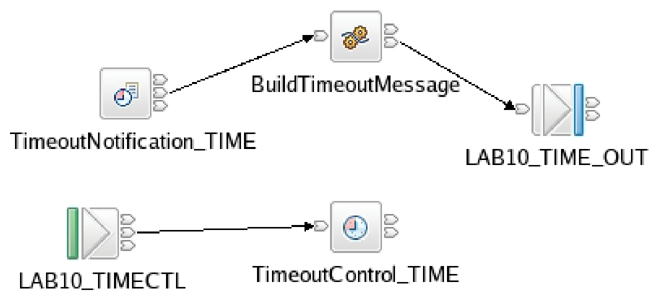
As a standalone node, `TimeoutNotification` nodes support simple, fixed interval timer events. For example, you can have a message flow executed every 30 seconds, every hour, or even every 17 years. Subsequent sections will explain what happens if the broker is not running when one of these timer events is supposed to happen.

There is little flexibility when only a `TimeoutNotification` node is used in a message flow. The node begins timings as soon as the containing message flow has been deployed, and the first timer event occurs as soon as the flow is deployed. You cannot stop the timer (short of stopping the containing flow itself), nor can you dynamically change the time interval. Also, if a `TimeoutNotification` node specifies that an event will occur ever hour, then it will occur at deployment and exactly one hour later. So, if a flow is deployed at 6:17am, then events will occur at 6:17 a.m., 7:17 a.m., 8:17 a.m., and so on.

TimeoutControl node

Including one or more `TimeoutControl` nodes in either the same or a different message flow (or even different execution groups) makes the patterns of generated timer events much more flexible. Like the `TimeoutNotification` node, the `TimeoutControl` node also has a unique identifier property (referred to as the UID). In this case however, despite the name of the property, it need not and should not be unique. Since the purpose of the `TimeoutControl` node is to dynamically set the characteristics of a `TimeoutNotification` node, the UID of the `TimeoutNotification` node should be the same as the UID of the controlling `TimeoutControl` node. Multiple `TimeoutControl` nodes can control the same `TimeoutNotification` node by specifying the same UID.

Figure 2. Including a TimeoutControl node. Note naming convention of adding UID (TIME) to node name



The `TimeoutControl` node has an `In` terminal to which you wire a source that provides a message tree with a certain structure (discussed in detail below). Elements of this tree include:

- Identifier
- Action
- Start Date
- Start Time
- Interval
- Count

For example, you can pass a message to the `TimeoutControl` node that directs a `TimeoutNotification` node to start the message flow at 17:50:08 on July 8, 2006, and then restart the flow every year for 50 times (50 years in this case).

You can wire the output of the `TimeoutNotification` node back into the `TimeoutControl` node. We will use this pattern later to set up timer events to happen in "bursts."

Comparing timer nodes to external flow-starting methods

These nodes let you start message flows based on timer events. There are of course other ways to start message flows:

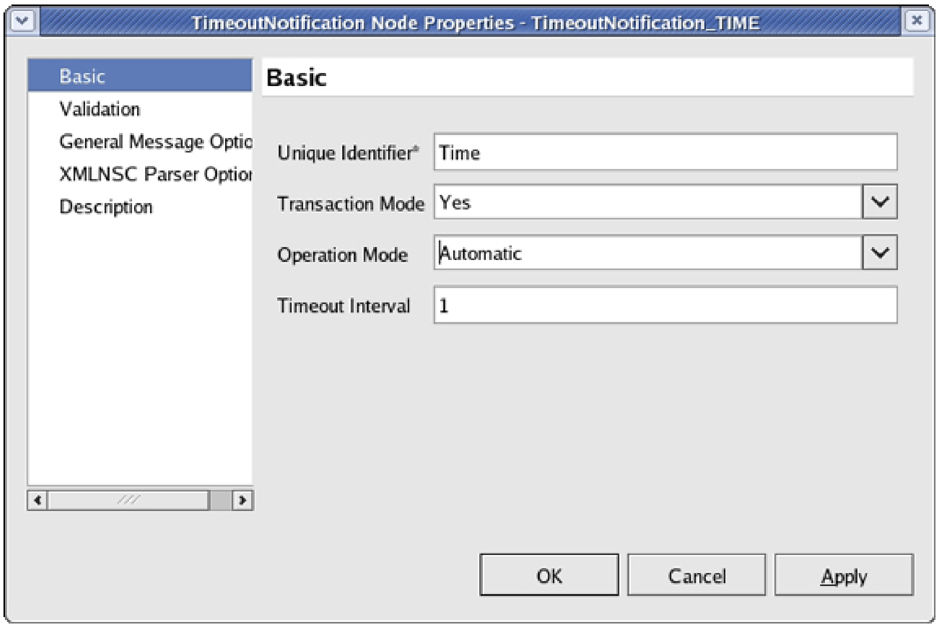
To make a flow run hourly on the hour, you can use a UNIX `cron` job, which can provide rich semantics to control job scheduling. The job would presumably use a utility such as `amqsput` to send a (dummy) message to the input queue of the message flow that you want to run.

While this approach works, it adds complexity to the system compared to using timer nodes. You must co-ordinate the creation of the `cron` job with the deployment of the message flows. If many flows have this requirement, maintenance of the crontab becomes burdensome. Another point of failure is introduced into the system, and you must have a mechanism to ensure that the `amqsput` utility actually worked and didn't generate a bad return code such as 2035. This approach makes problem determination more complex.

Using timeout nodes provides a flexible, self-contained mechanism to start a message flow. No external coordination with the broker is required beyond what is normally required to deploy a flow. You can use built-in, broker-native tracing mechanisms to detect the failure to set the timers. Although `cron` is usually available on UNIX, not all installations have it, and other platforms such as Windows® and z/OS® have different job scheduling mechanisms. Timeout nodes provide a consistent mechanism across all WebSphere Message Broker V6 platforms.

TimeoutNotification node details

Figure 3. Property box of the TimeoutNotification node



Basic

Unique Identifier* Time
Transaction Mode Yes
Operation Mode Automatic
Timeout Interval 1

The purpose of this node is to act as an alarm clock, initiating a message flow when the timer goes off. You can use it either standalone or you can pair it with one or more `TimeoutControl` nodes.

Figure 3 shows the property box of a `TimeoutNotification` node set to operate in standalone mode. Select standalone mode by setting the **Operation Mode** property to **Automatic**. Although there is no `TimeoutControl node`, you must still specify the UID (in our example it takes the value **Time**), and it still must be unique within the broker. The UID must be 1 to 12 characters -- the 12-character limit isn't enforced until you deploy the message flow.

In the standalone mode, set the value of **Transaction** to either **True** or **False** to indicate whether or not to start a transaction when the timer goes of. **Interval** is the time in seconds between timer events. The other property tabs don't affect the timer function.

As always, wire the `Catch` terminal of the `TimeoutNotification` node so that you avoid any unpleasant retry conditions. For more information, see "General Guidelines" below.

The message created on the `Out` terminal of this node looks like this:

```
Root:(
  (0x01000000):Properties = (
    (0x03000000):MessageSet      = ''
    (0x03000000):MessageType     = ''
    (0x03000000):MessageFormat   = ''
    (0x03000000):Encoding        = 546
    (0x03000000):CodedCharSetId  = 0
    (0x03000000):Transactional   = TRUE
    (0x03000000):Persistence     = FALSE
    (0x03000000):CreationTime    = TIMESTAMP '2005-08-11 17:34:54.802123'
    (0x03000000):ExpirationTime  = -1
    (0x03000000):Priority        = 0
    (0x03000000):ReplyIdentifier = X''
    (0x03000000):ReplyProtocol   = 'MQ'
    (0x03000000):Topic           = NULL
    (0x03000000):ContentType     = ''
  )
)
Environment:
LocalEnvironment:
(
  (0x01000000):TimeoutRequest = (
    (0x03000000):Action         = 'SET'
    (0x03000000):Identifier     = 'Time'
    (0x03000000):StartDate      = '2005-08-11'
    (0x03000000):StartTime      = '17:34:53.794321'
    (0x03000000):Count          = 2
    (0x03000000):Interval       = 1
    (0x03000000):IgnoreMissed   = TRUE
    (0x03000000):AllowOverwrite = TRUE
  )
)
```

The tree coming out of the node contains only a Properties and LocalEnvironment subtree. So if you plan on putting this message on a queue, you'll need to build a proper message for the `MQOutput` node (we'll do that later). StartDate and StartTime refer to when the timeoutNotification started. Count is how many times an event has occurred (in this case, it is the second event) and Properties.CreationTime gives the timestamp when the event actually happened.

What happens when the broker is not operational or the message flow with the `TimeoutNotification` node was stopped? Obviously, no timer events occur. But suppose the broker was down for 15 minutes and we set up a timer for once every minute, will the broker create 15 timer event messages when it restarts? The answer is no. For nodes in standalone mode, no state information is kept across restarts of the broker. The node restarts after the broker and flow restart and the Count field is reset to 1.

TimeoutControl node details

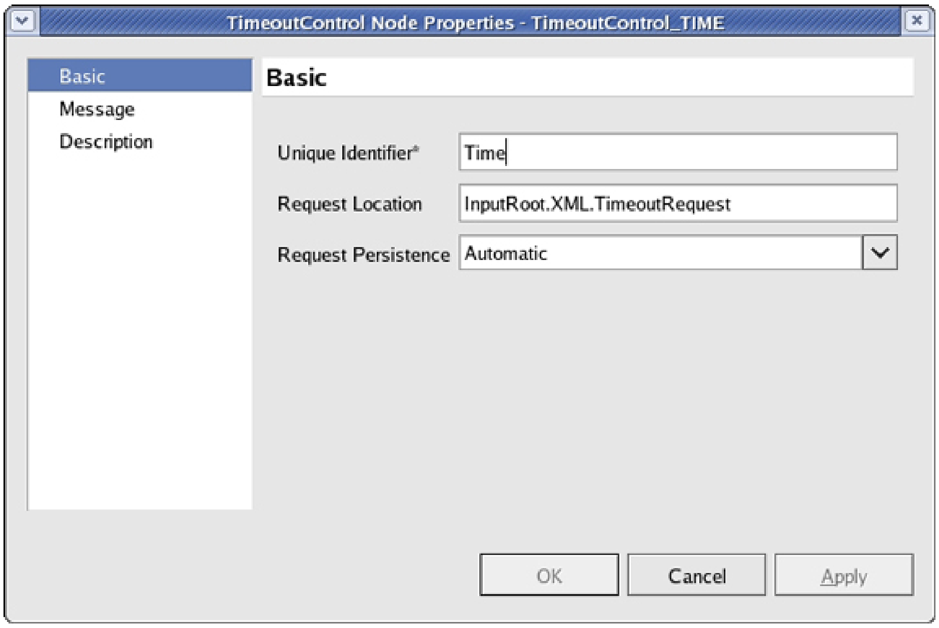
The purpose of a `TimeoutControl` node is to let you set of the parameters of a `TimeoutNotification` node.

You can use one or more `TimeoutControl` nodes in conjunction with a single `TimeoutNotification` node. In the simplest configuration, a single message flow could contain one `TimeoutControl` and its associated `TimeoutNotification` node. All the timeout nodes are associated via the UID specified in the properties of the node.

As shown in Figure 2, the `TimeoutControl` node has an `In` terminal, which you must wire to the source of a control message. Normally, this source is an `MQInput` node, but it can be any node that can produce the correct message.

As shown below in Figure 4, the UID specified here is the same as specified in Figure 3. Request Location is an optional parameter that defaults to `LocalEnvironment.TimeoutRequest` and specifies where the timer request subtree is located. Since we are sending an XML message to this node from an `MQInput` node, it is specified as `InputRoot.XML.TimeoutRequest`. Request persistence determines whether or not this timeout request survives broker restarts or the stopping of the execution group containing the `TimeoutNotification` node. **"Automatic** means that the persistence of the input message is used to determine if the timeout request survives broker restarts.

Figure 4. TimeoutControl node basic properties



Unique Identifier*: Time
Request Location: InputRoot.XML.TimeoutRequest
Request Persistence: Automatic

Here's the simplest input request message that you can send to the node:

```
<TimeoutRequest>
 <Identifier>TimeReq1</Identifier>
 <Action>SET</Action>
</TimeoutRequest>
```

The XML root name `TimeoutRequest` corresponds to the setting in the properties panel (`InputRoot.XML.TimeoutRequest`). The `Identifier` tag gives a unique identifier for this timeout request, which is not related to the UID of the nodes themselves. The `Action` tag indicates that we want to `SET` (create) a new timeout request. The other allowed action is `CANCEL`. When this message is sent into the flow, a single timer event occurs immediately. Once this single timer event occurs, the timeout request is considered completed.

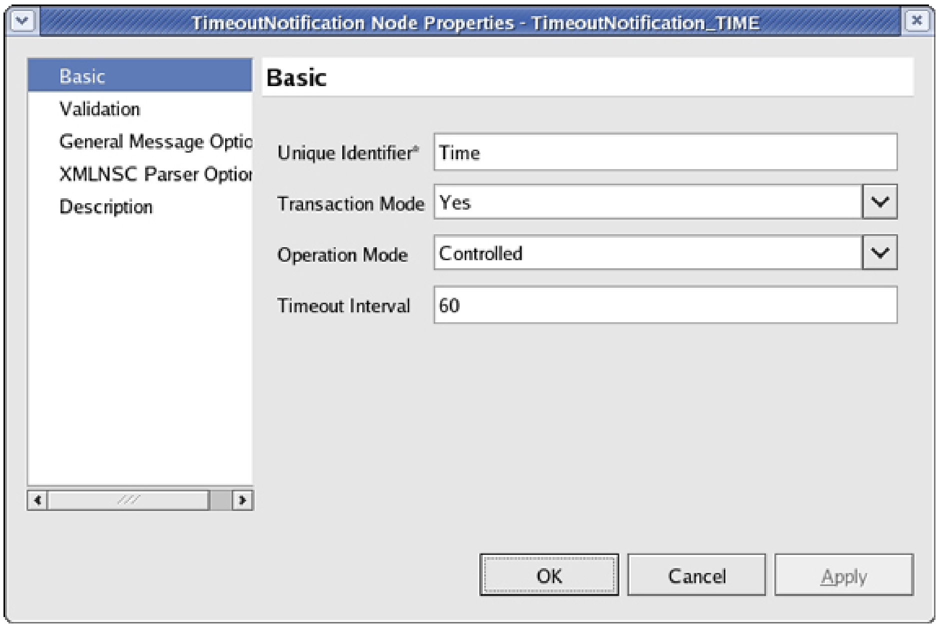
Of course, a message that immediately produces one timer event isn't very useful. A more practical message might be:

```
<TimeoutRequest>
 <Identifier>TimeReq1</Identifier>
 <Action>SET</Action>
 <Count>6</Count>
 <Interval>10</Interval>
</TimeoutRequest>
```

In this example, a timeout event happens immediately, and then recurs every 10 seconds (Interval) for a total of 6 times (Count). Count may take the special value "-1", which means forever.

Important: The `TimeoutNotification` node's Operation Mode properties must be changed from **Automatic** to **Controlled** for the `TimeoutControl` node to have any effect:

Figure 5. TimeoutNotification Node with Controlled Property set



Unique Identifier* Time
Transaction Mode Yes
Operation Mode Controlled
Timeout Interval 60

A fully specified control message looks like this: (See the InfoCenter on importing the IBM supplied schema (6.0.0\ibm\nodes\timeout\timeoutrequest.xsd) to create a message definition to map this message.)

```
<TimeoutRequest>
      <Identifier>TimeReq3</Identifier>
      <Action>SET</Action>
      <StartDate>2005-08-25</StartDate>
      <StartTime>NOW</StartTime>
      <Count>6</Count>
      <Interval>5</Interval>
      <IgnoreMissed>TRUE</IgnoreMissed>
      <AllowOverwrite>TRUE</AllowOverwrite>
</TimeoutRequest>
```

This example shows how you can request a timer to begin at a certain date and time. The StartDate is of the form "yyyy-mm-dd", and if the tag is omitted, defaults to the current date. The StartTime tag would indicate a local time to start generating timer events. It is of the form "hh:mm:ss.mmmmmm", with the "mmmmmm" portion being optional (however, the timer is only granular to 1 second intervals). The special value "NOW" may be specified for StartTime , which is the default value if the tag is not specified at all. The default value for StartDate is the special value "TODAY". StartDate can also be an element of type "Date" and StartTime can be an element of type "Time".

The IgnoreMissed flag indicates whether or not timer events that should have occurred while the broker was down (or the message flow containing the `TimeoutNotification` node was stopped) should be generated when the message flow starts again. The default value is "True", which means those events should be ignored (and not generated at restart). "False" means that those events should be generated at restart. However, in order for this to work as expected, you must set the Request Persistence on the `TimeoutControl` node to "Yes" (or you must send in a persistent message if its set to "automatic").

The AllowOverwrite flag indicates whether or not it is permissible to change the settings of a timer while that timer is still active. An active timer is one which is still producing timer events. So, for example, if you create a timer request with a name of "TimeReq1", a count of 10 and an interval of 6, that timer will remain active for fifty seconds (remember that the first timer event happens immediately). During this time, if a new control message is sent to the `TimeoutControl` node with the same request identifier ("TimeReq1") and the AllowOverwrite flag (on the first request) was set to "false", then the new message will raise an exception condition ('Timeout Set Identifier already used'). If however, the control message is send AFTER fifty seconds, and the timer is no longer active, it will be processed. Setting AllowOverwrite to true will always allow the second control message to be processed, regardless of the state of the timer.

It is important to note that the scope of the timer Identifier is the UID of the `TimeoutNotification` node. This means that if you have two `TimeoutNotification` nodes, one with a UID of "Time" and the other with a UID of "Tiempo", each can receive a control message with an Identifier of "TimeReq1" and these are distinct instances of a timer. (You might like to think of the timer identifier as "Time.TimeReq1" in one case and "Tiempo.TimeReq1" in the other.)

Lastly, a timer can be canceled by sending a control message with the "CANCEL" action. In this case, only the Action and Identifier need be specified in the control message. Canceling a timer that is not active will throw a 'Timeout Set Identifier not in store' condition.

Because the Operation Mode of the `TimeoutNotification` node is now "Controlled", the message that is sent to the Out terminal is different then when it was Automatic. In this case, the message tree carries the Root.MQMD and Root.XML of the message sent to the `TimeoutControl` node which started this timer. (We actually can control whether the entire control message is stored for propagation or only a portion of the message. See the "Stored Message Location" property of the `TimeoutControl` node.) This fact allows us to carry additional information from the message that started the timer to the flow that processes the timer event. Here's a sample of a message send to the "Out" terminal of the `TimeoutNotification` node (with some fields removed for clarity):

```
Root:
(
  (0x01000000):Properties = (
    (0x03000000):MessageSet      = ''
 ...
    (0x03000000):ContentType     = ''
  )
  (0x01000000):MQMD       = (
    (0x03000000):SourceQueue      = ''
    (0x03000000):Transactional    = TRUE
    (0x03000000):Encoding         = 546
    (0x03000000):CodedCharSetId   = 1208
    (0x03000000):Format           = 'MQSTR   '
 ...
    (0x03000000):OriginalLength   = -1
  )
  (0x01000010):XML        = (
    (0x01000000):TimeoutRequest = (
      (0x01000000):Identifier     = 'TimeReq3'
      (0x01000000):Action         = 'SET'
      (0x01000000):StartDate      = 'TODAY'
      (0x01000000):Count          = '60'
      (0x01000000):Interval       = '5'
      (0x01000000):IgnoreMissed   = 'FALSE'
      (0x01000000):AllowOverwrite = 'FALSE'
      (0x01000000):Redrive   = '60'
    )
  )
)
Environment:
LocalEnvironment:
(
  (0x01000000):TimeoutRequest = (
    (0x03000000):Action         = 'SET'
    (0x03000000):Identifier     = 'TimeReq3'
    (0x03000000):StartDate      = '2005-08-29'
    (0x03000000):StartTime      = '11:29:06.654868'
    (0x03000000):Count          = 1
    (0x03000000):Interval       = 5
    (0x03000000):IgnoreMissed   = FALSE
```

There is a field in the original message sent to the `TimeoutControl` node called "Redrive" (value =60). The field is a name that I made up, and therefore doesn't have any effect on the `TimeoutControl` node itself. However, it and any other field present on the control message is passed along from the `TimeoutNotification` node, which is a useful feature we'll take advantage of later. Note two other fields of interest: Root.XML.TimeoutRequest.Count (value = 60) and LocalEnvironment.TimeoutRequest.Count (value = 1). These two fields represent the original number of timer events requested and the current timer event number. This information let us determine if we are processing the last timer event for a given request.

Sending an MRM domain message to the node

Any properly formatted message subtree can be sent to the control node. As an example, you can send the control message in the MRM domain by specifying this for the "Request Location" **InputRoot.MRM** (There is a bug with the GA release. The CWF fields must be padded with SPACES (not NULLS). This will fixed with Fixpack FP1.)

A trace of the message flowing into the node might look like this:

```
Root:
((0x0100001B):MRM        = (
    (0x0300000B):Action         = 'SET'
    (0x0300000B):Identifier     = 'TIMEREQ89012'
    (0x0300000B):StartDate      = 'TODAY'
    (0x0300000B):StartTime      = 'NOW'
    (0x0300000B):Interval       = 60
    (0x0300000B):Count          = 10
    (0x0300000B):IgnoreMissed   = 'FALSE'
    (0x0300000B):AllowOverwrite = 'FALSE'
  )
```

Programming patterns

General guidelines

1. Give meaningful names to `TimeoutNotification` and `TimeoutControl` nodes. Suggestion: "TimeoutNotification_UID" and "TimeoutControl_UID", where UID is the 1 to 12 character unique identifier of the node. If there is a need for more than one `TimeoutNotification` node in the flow, then append "_1, _2" etc to the name.
2. Always wire the Catch terminal of the `TimeoutNotification` node. The `TimeoutControl` node has no Catch terminal, so errors are either sent to the Failure Terminal or thrown upstream. Provision should be made for handling these errors.
3. Unless there is a compelling need, the Request Persistence on the `TimeoutControl` node should be explicitly set to "Yes" or "No". Allowing the setting to be "Automatic" makes the system behavior too dependent on the input message's persistence.
4. Wire a `Reset Content Descriptor` (RCD) node to the Catch node of a `TimeoutNotification` node. This prevents an exception being thrown by a subsequent node (such as a `Trace` node) in the case where a message stored by a `TimeoutControl` node cannot be parsed by the `TimeoutNotification` node and wouldn't be able to be parsed by a `Trace` node either. The RCD node should reset the domain to BLOB. (As of March 2006, there is a defect in the way RCD's nodes handle the output from the `TimeoutNotification` node in "Automatic Mode". This will be fixed in a future Fixpack.)

Scenario 1. Triggering a flow every "n" seconds forever

Requirements

Attribute	Value
Timer event messages are required every "n" seconds	Yes
Timer events should be generated whenever the containing message flow is running	Yes
If the message flow (or broker) is not running, missed timer events should be generated at startup	No
The exact time (wall clock time) of the timer event is important	No

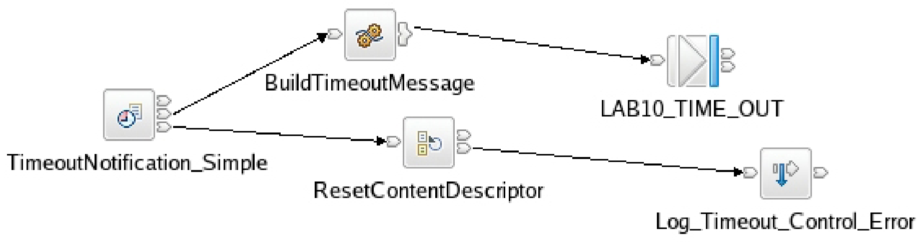
Possible scenarios

This scenario would occur most likely if we needed a broker flow to run for a maintenance purpose, such as processing or clearing rows that have accumulated in a data base table. Generally, we don't care exactly when these maintenance procedures occur, as long as they occur on a periodic basis (for example: "clear the database once an hour").

Programming pattern

Given these requirements, the best practice solution would be to code a single "stand-alone" `TimeoutNotification` node as the head of a flow:

Figure 6. Scenario 1 Message Flow



A `Reset Content Descriptor` (RCD) node has been inserted after the Catch node. This prevents an exception being thrown by the `Trace` node in the case where a message stored by a `TimeoutControl` node cannot be parsed by the `TimeoutNotification` node and wouldn't be able to be parsed by the `Trace` node either. The RCD node resets the domain to BLOB.

As stated earlier, the `TimeoutNotification` node in "Automatic" mode does not produce a message tree suitable for sending to an `MQOuput` node (for example, there is no MQMD subtree). The following ESQL code can be used to create an appropriate message (and would be coded in the "BuildTimeoutMessage" node above).

```
CREATE FUNCTION Main() RETURNS BOOLEAN
 BEGIN
   CALL CopyMessageHeaders();
        -- Capture the request as an output message
        SET "OutputRoot"."Properties"."MessageDomain" = 'XML';
        SET "OutputRoot"."MQMD"."Format" = MQFMT_STRING;
        SET "OutputRoot"."XML"."TimeoutRequest" = "InputLocalEnvironment"."TimeoutRequest";
   RETURN TRUE;
 END;
```

Scenario 2: Triggering a flow every "n" seconds at a certain time of day

Requirements

Attribute	Value
Timer event messages are required every "n" seconds	Yes
Timer events are required to commence at a specific date-time.	No, they should commence when the broker starts
Once started, "m" events should occur.	Either
The exact time (wall clock time) of the timer event is important	Yes
If the message flow (or broker) is not running, missed timer events should be generated at startup	Either

Possible scenarios

The significant difference between this and the prior scenario is that we want to have the timer events occur at a specific time, such as "once an hour at 10 minutes past the hour". "m" could be either a fixed number of events or an infinite number.

Programming pattern

Since the timer events must occur at a specific time (for example, at the top of the hour, like 1:00 p.m.), it is necessary to use a `TimeoutControl` node. There are two approaches to using this node.

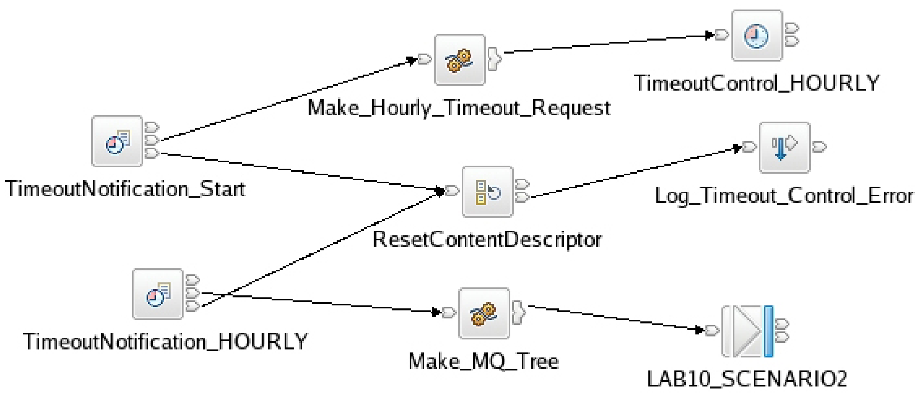
The first approach is to simply wire an `MQInput` node to the `TimeoutControl` node and then send a control message to the queue associated with the `MQInput` node. This message would be an example of the message that would need to be sent:

```
<TimeoutRequest>
      <Identifier>HourlyReq</Identifier>
      <Action>SET</Action>
      <StartDate>TODAY</StartDate>
      <StartTime>13:00:00</StartTime>
      <Count>-1</Count>
      <Interval>3600</Interval>
</TimeoutRequest>
```

If the Request Persistence is set to "Yes", then this message will only need to be sent once. It will need to be sent immediately after the message flow containing the `TimeoutNotification` node is deployed. There is an issue here, and that is setting of the "StartTime" field. The value must be set to the next hour (or whatever specific time is required for the application). Therefore, a customized application must be written to create a message with the proper value of "StartTime". This is clearly undesirable.

The second solution, which is the best practice, is to use a `TimeoutNotification` node in "Automatic" mode to generate the control message for us. Figure 7 shows a flow that automatically generates a control message for the `TimeoutControl` node from a `TimeoutNotification` node in "Automatic" mode.

Figure 7. A message flow to automatically generate the control message


Make_Hourly_Timeout_Request
TimeoutControl_HOURLY
TimeoutNotification_Start
Log_Timeout_Control_Error
ResetContentDescriptor
TimeoutNotification_HOURLY
Make_MQ_Tree
LAB10_SCENARIO2

When this flow is initially deployed (and on subsequent deploys), the node "TimeoutNotification_Start" immediately creates a message which is propagated to its "Out" terminal. The properties of the node are as follows:

Property	Value
Unique Identifier	Start
Transaction Mode	Yes
Operation Mode	Automatic
Timeout Interval	3300

Timeout Interval is set to 3300 seconds (55 minutes) to allow for the condition where the flow is deployed so close to the top of the hour that the message does not arrive at the `TimeoutControl` node before the top of the hour. By setting the Timeout Interval to 55 minutes, we are guaranteed that a control message will be setup for the next hour.

From this message, we use a compute node ("Make_Hourly_TimeoutRequest") to create a control message which asks for timer events to happen at the top of the hour. Remember to allow the node to update the "Message + LocalEnvironment" tree in its property box (Compute mode). The ESQL code for this node is listed below:

```
CREATE COMPUTE MODULE Make_Hourly_TimeoutRequest
 CREATE FUNCTION Main() RETURNS BOOLEAN
 BEGIN
 declare nexthour_c character;
 declare nexthour_i integer;
     declare nexthour_t time;
     declare nextday_d date;
      CALL CopyEntireMessage_and_Environment();
       -- grab just the hour out of the current time ...
        set nexthour_c = cast(CURRENT_TIME as character format 'HH');
 set nexthour_i = cast(nexthour_c as integer);
 -- set the timer for the next hour ....
 set nexthour_i = nexthour_i + 1;
 -- if its already after 11pm, we have to bump the date as well.
 if (nexthour_i > 23) then
  -- select midnight
  set nexthour_i = 0;
  -- select tomorrow's date
  set nextday_d = cast(InputLocalEnvironment.TimeoutRequest.StartDate as date);
  set nextday_d = nextday_d + INTERVAL '1' day;
  set OutputLocalEnvironment.TimeoutRequest.StartDate = nextday_d;
 end if;
 set nexthour_t = cast(nexthour_i, 0, 0 as time);
 set OutputLocalEnvironment.TimeoutRequest.StartTime = cast(nexthour_t as time);
 -- forever
 set OutputLocalEnvironment.TimeoutRequest.Count = -1;
 -- once an hour
     set OutputLocalEnvironment.TimeoutRequest.Interval = 3600;
     set OutputLocalEnvironment.TimeoutRequest.AllowOverwrite = TRUE;
 RETURN TRUE;
 END;

 CREATE PROCEDURE CopyEntireMessage_and_Environment() BEGIN
  SET OutputRoot = InputRoot;
  set OutputLocalEnvironment = InputLocalEnvironment;
 END;
END MODULE;
```

The properties of "TimeoutControl_HOURLY" are:

Property	Value
Unique Identifier	HOURLY
Request Location	(blank)
Request Persistence	No
Message Domain	XML

Since the "TimeoutNotification_Start" node will always generate an event after the broker restarts (or a message flow is restarted), there is no need to persist this timer. The default value of "InputLocalEnvironment.TimeoutRequest" is taken for Request Location.

The properties of "TimeoutNotification_HOURLY" are:

Property	Value
Unique Identifier	HOURLY
Transaction Mode	Yes
Operation Mode	Controlled
Timeout Interval	1 (ignored since this is a "Controlled" node)

With an "Operation Mode" of "Controlled", the value of "Timeout Interval" is ignored.

The "Make_MQ_tree" compute node is needed to construct a proper tree for the `MQOutput` node (see scenario 1). Like the previous example which used the `TimeoutNotification` node, there is no MQMD or XML subtree propagated from the `TimeoutControl` node, so we need to set the domain and create the MQMD tree:

```
CREATE FUNCTION Main() RETURNS BOOLEAN
    BEGIN
     CALL CopyMessageHeaders();
 -- Capture the request as an output message
 SET "OutputRoot"."Properties"."MessageDomain" = 'XML';
 SET "OutputRoot"."MQMD"."Format" = MQFMT_STRING;
 SET "OutputRoot"."XML"."TimeoutRequest" = "InputLocalEnvironment"."TimeoutRequest";
 RETURN TRUE;
    END;
```

Scenario 3. Trigger a flow ever "n" seconds for "m" times and then wait "z" seconds before next set

Requirements

Attribute	Value
Timer event messages are required every "n" seconds	Yes, after a burst of event occur, we want to wait for a specific period of time before the next burst.
Timer events are required to commence at a specific date-time.	Yes
Once started, "m" events should occur.	Yes
If the message flow (or broker) is not running, missed timer events should be generated at startup	Either

Possible scenarios

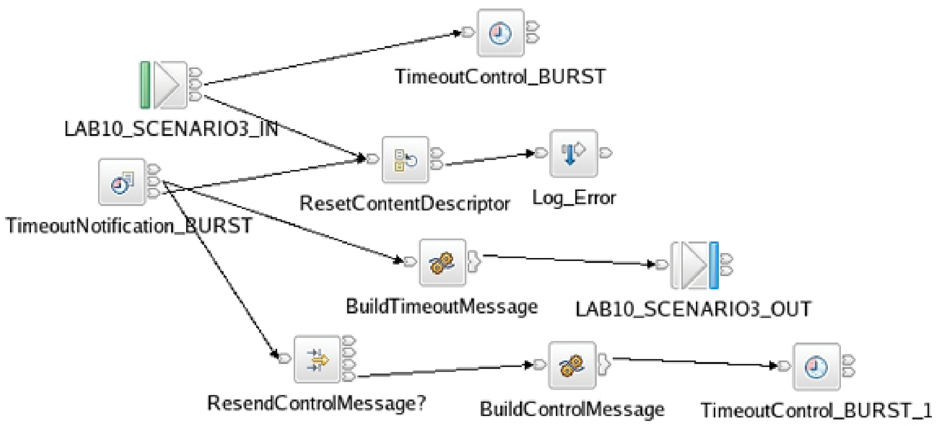
In this scenario, we want a burst of timer events for a period of time, followed by no events for a period of time, followed by another burst of events. So, for example, suppose once an hour we want to get timer events every minute for 10 minutes, and then 50 minutes where no timer events occur.

Programming pattern

Since the timer events must occur at a specific time (for example, at the top of the hour, like 1:00 p.m.) and/or there must be a specific number of them in the "burst," it is necessary to use a `TimeoutControl` node. As in Scenario 2, the `TimeoutControl` node can be set by an external message arriving or a `TimeoutNotification` node. For this scenario, we'll use a message as input.

As an example, the initial control message might have a Count of 10 and an Interval of 60. This would give us the first burst of timer events. After the last timer event occurs, we need to send another control message back into a `TimeoutControl` node, with either the same or different values (and possibly setting StartDate or StartTime). Here's a flow that can accomplish that:

Figure 8. Scenario 3 Message Flow



Unlike the previous scenario, we use an `MQInput` node to send the first timer control message. Here's the XML the input message contains:

```
<TimeoutRequest>
      <Identifier>Time0</Identifier>
      <Action>SET</Action>
      <Count>3</Count>
      <Interval>10</Interval>
      <BurstInterval>60</BurstInterval>
</TimeoutRequest>
```

Since we don't specify a StartDate or StartTime, they default to TODAY and NOW. This message asks for 3 messages 10 seconds apart. The field "BurstTime" is not a field recognized by the Timeout nodes. It's a field we will use later to set the interval between the bursts. As stated earlier, the `TimeoutControl` node will save this field and we will receive it in the message tree passed from the `TimeoutNotification` node.

The `MQInput` node sends the control message into the `TimeoutControl` node "TimeoutControl_BURST", which has the following properties:

Property	Value
Unique Identifier	BURST
Request Location	InputRoot.XML.TimeoutRequest
Request Persistence	Yes
Message Domain	XML

The "TimeoutNotification_BURST" Node has these properties:

Property	Value
Unique Identifier	BURST
Transaction Mode	Yes
Operation Mode	Controlled
Timeout Interval	1 (ignored since this is a "Controlled" node)

When the initial control message is sent to this node, the "TimeoutNotify_BURST" node will create timer events at the specified intervals. All these timer event messages are sent on to the "BuildTimeoutMessage" compute node for further processing. All we need to do in this node is to move the timeout message to the XML domain:

```
CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
  CALL CopyMessageHeaders();
  SET "OutputRoot"."Properties"."MessageDomain" = 'XML';
  SET "OutputRoot"."MQMD"."Format" = MQFMT_STRING;
         SET "OutputRoot"."XML"."TimeoutRequest" = "InputLocalEnvironment"."TimeoutRequest";
 RETURN TRUE;
   END;
```

At the end of the "burst" of timer messages, we want to send another timer control message to setup the next set of timer bursts. First, we must detect that we have received the final timer event in the burst. We do this by using a `Filter` node and testing the value of "LocalEnvironment.TimeoutRequest.Count" (which is set by the `TimeoutNotification` node and indicates which timer event in the burst this represents) against Root.XML.TimeoutRequest.Count (which was set by the initial control message and by default is propagated along with the timer event message. The subtree propagated by the TimeoutNotification node is controlled by the "Stored Message Location" property of the `TimeoutControl` node. By default, the entire control message is propagated.) Here's the code for the filter node "ResetControlMessage?":

```
CREATE FILTER MODULE ResetControlMessageQ
    CREATE FUNCTION Main() RETURNS BOOLEAN
 BEGIN
 if LocalEnvironment.TimeoutRequest.Count = Root.XML.TimeoutRequest.Count then
    RETURN TRUE;
 else
    RETURN FALSE;
 end if;
     END;
END MODULE;
```

If we do detect that this is the final timer event in the burst, then we must build a new control message to setup for the next burst. Below is an incomplete code segment for taking the timer event message (which is stored in LocalEnvironment), and fixing up the StartTime by adding "BurstInterval" to it. Remember that StartTime represents the time of the first timer event, not the last, so adding the value of "BurstInterval" correctly sets the time for the next event.

```
CREATE FUNCTION Main() RETURNS BOOLEAN
 BEGIN
     declare inter interval;
     -- copy Root and LocalEnvironment
     set OutputRoot = InputRoot;
     set OutputLocalEnvironment = InputLocalEnvironment;

     -- make BurstInterval an "interval" type
      set inter = cast(InputRoot.XML.TimeoutRequest.BurstInterval as interval second);
      -- and fix up the time to be a "burstInterval" from the last startTime ....
     set OutputLocalEnvironment.TimeoutRequest.StartTime =
      cast(InputLocalEnvironment.TimeoutRequest.StartTime as time) + inter;

       -- if we go over midnight, the date will also need to be fixed
           -- this is left as an exercise for the student
 RETURN TRUE;
 END;
```

Lastly, the second TimeoutControl node "TimeoutControl_Burst_1" has the following properties:

Property	Value
Unique Identifier	BURST
Request Location	InputLocalEnvironment.TimeoutRequest
Request Persistence	Yes
Message Domain	XML

On a final note, we could have used a single `TimeoutControl` node in this flow. If that was done, it would have been necessary to change the compute node to place the control message in the same location as it is when received from the `MQInput` node (that is, it would need to be in

Root.XML). Two nodes where chosen here to make the wiring a little cleaner and demonstrate the use of two control nodes for a single notification node.

Scenario 4. Timer events at a given time of day Monday through Friday

Requirements

Attribute	Value
Timer event messages are required periodically at a certain time of day	Yes
Timer events are required to commence at a specific date-time.	Yes
Certain Days of the week should be excluded.	Yes
If the message flow (or broker) is not running, missed timer events should be generated at startup	Either

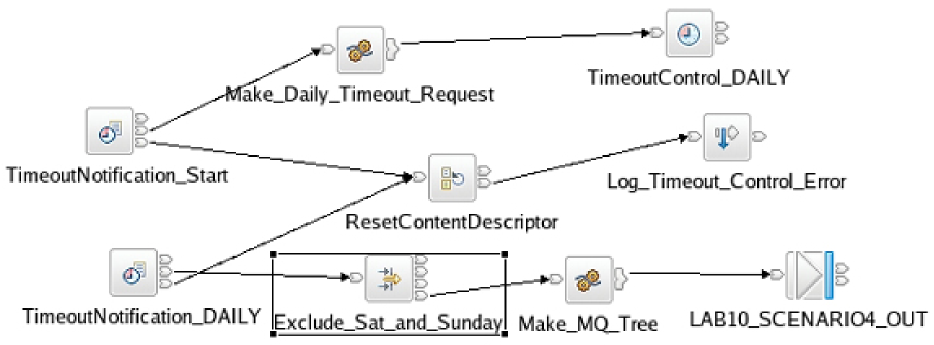
Possible scenarios

This is fundamentally "Scenario 2" with the desire to filter out certain timer events. As an example, we might wish to exclude Saturdays and Sundays.

Programming pattern

We will use the "Scenario 2" message flow with the addition of a filter node:

Figure 9. Scenario 4 Message Flow


Make_Daily_Timeout_Request
TimeoutControl_DAILY
TimeoutNotification_Start
Log_Timeout_Control_Error
ResetContentDescriptor
TimeoutNotification_DAILY
Exclude_Sat_and_Sunday
Make_MQ_Tree
LAB10_SCENARIO4_OUT

To set up the filter node, we make the following test:

```
CREATE FILTER MODULE Exclude_Sat_and_Sunday_Filter
 CREATE FUNCTION Main() RETURNS BOOLEAN
 BEGIN
  declare day_of_week_i integer;
  declare day_of_week_c character;
  -- 'e' is day of week :
  -- Monday = 1, Friday = 5, Sat = 6, Sun = 7
  set day_of_week_c = cast(current_date as char format 'e');
  set day_of_week_i = cast(day_of_week_c as integer);
  if (day_of_week_i > 5) then
   RETURN FALSE;
  else
   return true;
  end if;
 END;
END MODULE;
```

Scenario 5. Using timeout nodes to introduce a timed delay into a flow

Requirement

Attribute	Value
There is a requirement to delay a message flow for a period of time	Yes

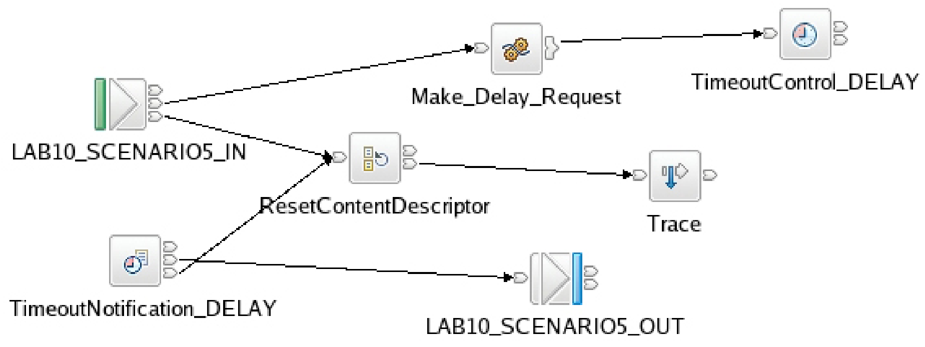
Possible scenarios

In this scenario, we have a message flow where we want to delay the flow for a period of time. This would typically be where we need the flow to wait for an asynchronous event to occur (which takes a fixed period of time). The flow would start processing a message, wait for a period of time (for example, 60 seconds) and then continue processing the message after that time period.

Programming pattern

In this scenario, we wire a `TimeoutControl` and a `TimeoutNotification` node into the flow and intentionally propagate the message tree thru the Timeout nodes themselves. This sample flow shows a solution:

Figure 10. Scenario 5 Message Flow



The ESQL in the "Make_Delay_Request" node is:

```
declare myCounter shared integer 0;

CREATE COMPUTE MODULE Make_Delay_Request_Compute
 CREATE FUNCTION Main() RETURNS BOOLEAN
 BEGIN

 CALL CopyEntireMessage();
 begin atomic
           set OutputLocalEnvironment.TimeoutRequest.Identifier =
        'TimeReq' || cast (myCounter as character) || SQL.ExecutionGroupLabel;
           set myCounter = myCounter + 1;
        end;
        set OutputLocalEnvironment.TimeoutRequest.Action = 'SET';
 set OutputLocalEnvironment.TimeoutRequest.IgnoreMissed = FALSE;
        set OutputLocalEnvironment.TimeoutRequest.StartTime =
        cast(CURRENT_TIMESTAMP + INTERVAL '60' second as char format 'HH:mm:ss');
 RETURN TRUE;
 END;
END MDULE;
```

We don't really care what the contents of the message tree are at this point, our main concern is to create a "LocalEnvironment.TimeoutRequest" subtree with the "Identifier", "Action" and "StartTime" values.

This message flow differs in one important respect from flows that don't use this technique. In other flows, if there is only one thread of execution allowed and multiple messages are on the input queue, the first message is processed to completion before the second message is read. In this flow, the first message is processed until the `TimeoutControl` node is complete. Then the second message is processed. So it's possible that many messages are processed off the input queue before the first message is placed onto the output queue. This means that its likely there will be many timers active at once, and for this reason each timer must have a unique Identifier. To create a counter that can be shared across execution threads, we use the new "shared" data type to declare the "myCounter" integer. This integer is initialized when the flow is deployed (or restart of the broker or message flow). (As of March 2006, there is a bug that resets this variable the first time the timer on the flow timers out.) Since this counter is NOT shared across execution groups, it is necessary to also append the execution group name to the counter to ensure uniqueness (remembering that timer events are shared across execution groups). To ensure integrity across multiple instances of this flow, the use and updating of the variable is enclosed within a "begin atomic...end" block. The "IgnoreMissed" attribute is set on the request because it is likely that we would not want to lose messages because the broker was not running.

The `TimeoutControl` node "TimeoutControl_DELAY" has the following properties:

Property	Value
Unique Identifier	DELAY
Request Location	Blank, which defaults to InputLocalEnvironment.TimeoutRequest
Request Persistence	Yes
Stored Message Location	InputRoot
Message Domain	BLOB

The last two properties, "Stored Message Location" and "Message Domain" are important because they indicate where to find the message tree that we want to propagate out of the `TimeoutNotification` node. In this case, we want to propagate the entire original message tree of the InputRoot (the original message read by the `MQInput` node). This saves both the MQMD and the message payload (we want to save the MQMD for the `MQOutput` node later on). Since we don't care about the structure of the message, we set the domain to "BLOB".

The `TimeoutNotify` node "TimeoutNotify_Delay" has the following properties:

Property	Value
Unique Identifier	Delay
Transaction Mode	Yes
Operation Mode	Controlled
Timeout Interval	1 (ignored since this is a "Controlled" node)

Dequeuing is single threaded, even if the message flow itself is configured for multiple instances. We therefore directly wire the `TimeoutNotification` node to an `MQOutput` node which could feed multiple instances of a subsequent flow where the business process would be executed.

Scenario 6. Using timeout nodes to introduce a timed delay into a flow as part of exception processing

Requirements

Attribute	Value
There is a requirement to delay a message flow for a period of time after an exception occurs.	Yes

Possible scenarios

In this scenario, we have a message flow where we want to delay the flow for a period of time after an exception has occurred. This is similar to Scenario 5. In this scenario, we have reason to believe that the condition which caused the exception is transitory, and it makes sense to retry the transaction after a period of time. Possible examples would include "queue full" and "database connection not available" (due to the DB being down).

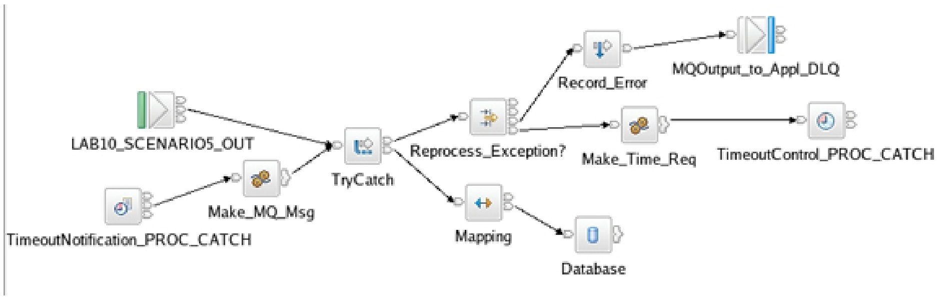
Programming pattern

This is really a continuation of Scenario 5. As Scenario 6 shows, we connect the `TimeoutNotification` node to a catch terminal. A `Filter` node is used to select only those exception conditions that we want to reprocess after a delay (the others are just sent to a `Trace` node). The filter should also ensure that we are not in a "poisoned message" situation by ensuring that the message has been reprocessed less than "n" times. (This could be done by carrying a counter in the timer request message itself and incrementing each time thru, see the discussion of "Redrive" previously).

As in Scenario 5, we need to make a timer request message before passing the original message to a `TimeoutControl` node.

Since we are feeding both the original MQ message (from the `MQInput` node) and the tree from the `TimeoutNotification` node into the "TryCatch" node, we must also ensure that the trees from the two sources are identical. We use a "Make_MQ_Msg" compute node to accomplish this.

Figure 11. Scenario 6 Message Flow



For a complete solution for retrying failed message flows, see Generic message retry and requeue with WebSphere Message Broker V6.

Conclusion

WebSphere Message Broker V6 introduces several powerful new ways to initiate message flows using the `TimeoutNotification` and `TimeoutControl` nodes. This paper has explained best practices for using these nodes effectively in several different scenarios.

Related topics

- WebSphere Message Broker documentation in IBM Knowledge Center
- WebSphere Message Broker documentation library
- WebSphere Message Broker product page
- Generic message retry and requeue with WebSphere Message Broker V6
- Most popular trial downloads