developerWorks®

Extending aggregation and collection of messages in WebSphere Message Broker

Peter Broadhurst
Patrick Marie

April 17, 2013

This article shows you how to use WebSphere Message Broker message aggregation and collection to manage batch processing according to dynamic rules.

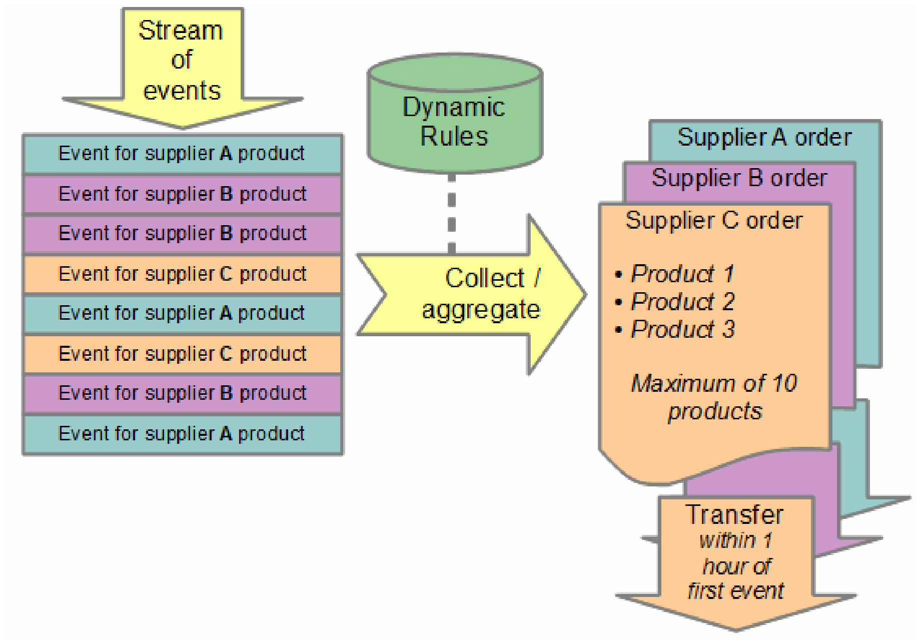
Introduction

This article describes a scenario involving collecting messages into batches for processing, and shows you how to extend the built-in aggregation and collection features in IBM® WebSphere® Message Broker (hereafter called Message Broker). An example shows you how to extend the built-in features to dynamically configure the batching based on rules stored in a database, including rules for timeout, maximum batch size, and stranded message protection. The example and builds on the logical message groups feature of WebSphere MQ (hereafter called MQ), which is part of the Message Broker product.

The scenario

The scenario is a stock-management system to consolidate orders and pass them to individual suppliers:

Figure 1. The scenario

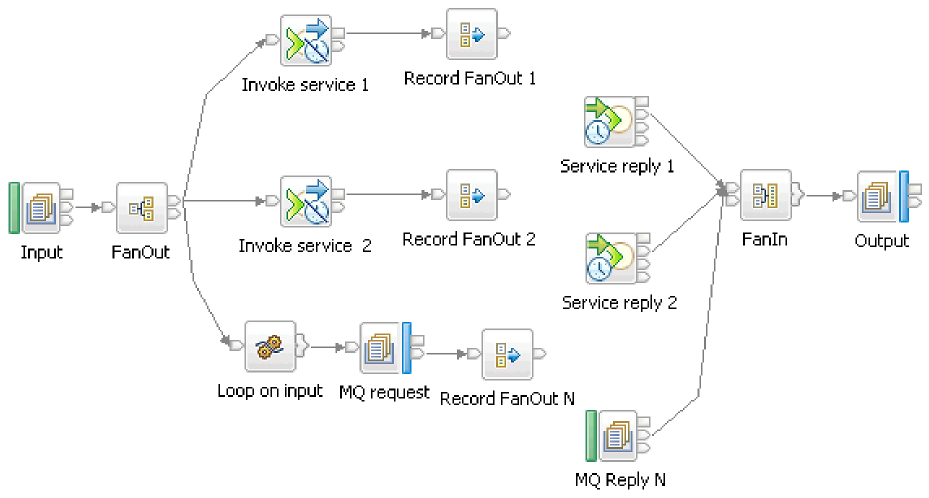

Stream of events
Event for supplier A product
Event for supplier B product
Event for supplier B product
Event for supplier C product
Event for supplier A product
Event for supplier C product
Event for supplier B product
Event for supplier A product
Dynamic Rules
Collect / aggregate
Supplier A order
Supplier B order
Supplier C order
• Product 1
• Product 2
• Product 3
Maximum of 10 products
Transfer within 1 hour of first event

- Events are generated when the stock level of a product drops below a certain threshold.
- Each event requires a product to be ordered from a particular supplier, and different events may need products from different suppliers.
- The suppliers' systems require files to be transferred to them containing a list of product orders. It is inefficient and unreliable to transfer a separate file for each product. The following restrictions also apply:
  - Just-in-time operation means that if sending an order to a supplier is delayed by more than a certain threshold, then it has a business impact. The rules for how long orders for particular products can be delayed are complex, and change regularly.
  - Each supplier has limits on how many product orders can be contained in an individual file.

Built-in aggregation nodes

Message Broker provides several aggregation nodes: AggregateControl, AggregateRequest, and AggregateReply. At first glance these nodes might seem a good fit for the scenario, but they are designed to solve a slightly different problem -- fan-out/fan-in logic. For example, a process may need to issue a set of actions, such as web service invocations against a set of back-end services, and then wait for the results once all of the actions are complete. Figure 2 shows the kind of fan-out/fan-in logic that is ideally suited to the built-in Message Broker aggregation nodes:

Figure 2. Fan-out/fan-in logic with asynchronous web service and MQ requests

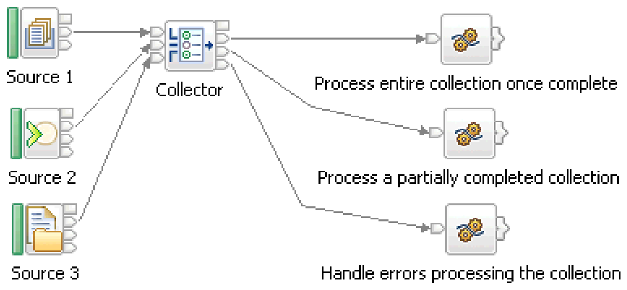


Using the aggregation nodes to fulfil the requirements in this article not work well, because the flow would have to block waiting for all of the messages for a particular group to arrive before it could finish the fan-out phase.

Built-in collection nodes

The scenario here is actually a much better fit for the Message Broker collection features and Collector node, which lets you group messages based on specific criteria. You can collect messages from multiple sources, with a different set of criteria applied against each source to build up the collection. Figure 3 shows the flexibility of the Collector node:

Figure 3. Collector node collecting data from multiple sources


Source 1
Collector
Process entire collection once complete
Source 2
Process a partially completed collection
Source 3
Handle errors processing the collection

Here are the criteria that can be applied to each source to build up the collection:

- A target number of messages to collect
- A timeout period to wait for messages to arrive
- A correlation pattern to match against a correlation path in each message

While the Collector node is highly flexible, and its function seems a good fit for the scenario, the following challenges remain:

- The timeout period is fixed for the Collector node, so a different Collector node is required for every timeout value, and changing a timeout period would require redeployment.

- Acceptance into a particular collection is based on a single criteria, and in the scenario, the rules defining which collection a particular low-stock event needs to be batched into may require more complex logic.

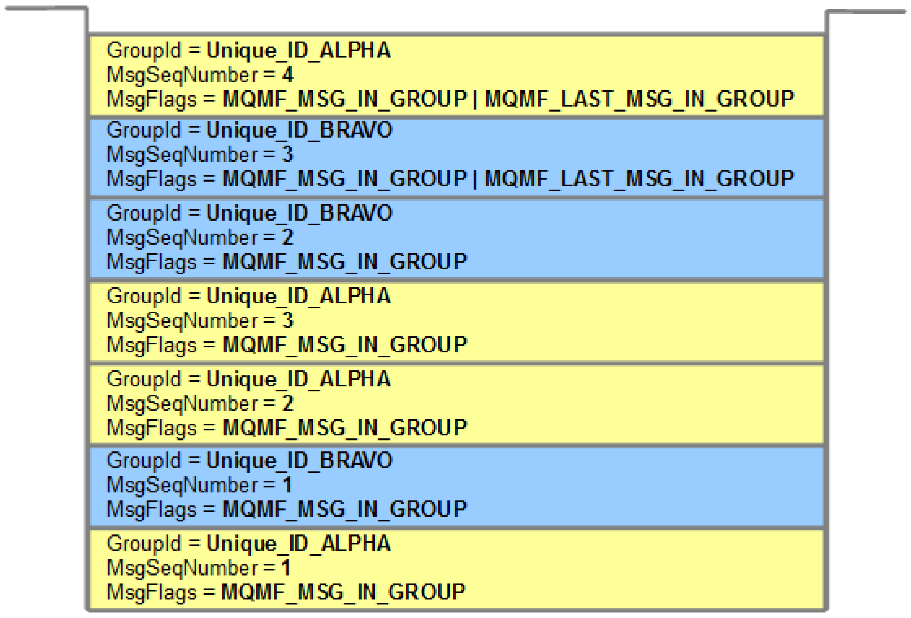
MQ logical groups

When the built-in aggregation or collection nodes do not provide enough flexibility to solve a problem, such as in our dynamic rule-based scenario, then you can use MQ *logical groups* to aggregate messages.

A logical group is a set of MQ messages with the following attributes:

- The same unique identifier set in the `GroupId` field of the `MQMD`. The unique identifier is usually generated by MQ.
- A `MsgSeqNumber` set in the `MQMD` of each message, indicating its position in the group starting from 1.
- Appropriate message grouping flags set in the `MsgFlags` field of the `MQMD` of each message:
    - The `MQMF_MSG_IN_GROUP` set on every message.
    - The `MQMF_LAST_MSG_IN_GROUP` set on the last message in the group.

Figure 4. An MQ queue containing two logical groups of messages


GroupId = Unique_ID_ALPHA
MsgSeqNumber = 4
MsgFlags = MQMF_MSG_IN_GROUP | MQMF_LAST_MSG_IN_GROUP
GroupId = Unique_ID_BRAVO
MsgSeqNumber = 3
MsgFlags = MQMF_MSG_IN_GROUP | MQMF_LAST_MSG_IN_GROUP
GroupId = Unique_ID_BRAVO
MsgSeqNumber = 2
MsgFlags = MQMF_MSG_IN_GROUP
GroupId = Unique_ID_ALPHA
MsgSeqNumber = 3
MsgFlags = MQMF_MSG_IN_GROUP
GroupId = Unique_ID_ALPHA
MsgSeqNumber = 2
MsgFlags = MQMF_MSG_IN_GROUP
GroupId = Unique_ID_BRAVO
MsgSeqNumber = 1
MsgFlags = MQMF_MSG_IN_GROUP
GroupId = Unique_ID_ALPHA
MsgSeqNumber = 1
MsgFlags = MQMF_MSG_IN_GROUP

Messages in MQ logical groups can be put onto a queue one message at a time, and the group remains incomplete until the last message is put on the queue with the `MQMF_LAST_MSG_IN_GROUP` flag set. Many groups can very efficiently exist on the same queue at the same time, and groups can remain incomplete for minutes, hours. or days. Therefore you can use MQ logical groups in Message Broker to aggregate messages in a flexible and dynamic way. You have to remember which unique group identifiers are used, and an example below will show you how to achieve this, as well as how to ensure that groups do not become stranded in error or restart scenarios.

Processing MQ logical groups in Message Broker

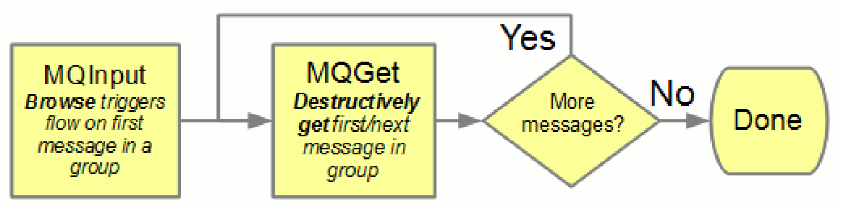
Building MQ logical groups of messages is only half the story -- after the groups are built, you need to trigger the logic to process the entire logical group.

Logic built in Message Broker using the `MQInput` and `MQGet` nodes can request that MQ hide messages in a logical group until all messages in a group are available. To do so, simply specify the `MQGMO_LOGICAL_ORDER` and `MQGMO_ALL_MESSAGES_AVAILABLE` flags when browsing or getting the messages.

After a handle to a queue has been used to receive the first message in a logical group, MQ delivers the subsequent messages in that group, in order, to that same handle, as long as `MQGMO_LOGICAL_ORDER` is specified on each Get. As long as the same handle to MQ is used to get all of the messages, MQ will deliver all of the messages in the group in the right order.

In Message Broker, an `MQGet` node uses a different handle to receive messages to the `MQInput` node in that same flow. Therefore a flow that consumes a logical group must be triggered by an `MQInput` that only browses the first message in a group and obtains the `GroupId`. The flow should then destructively get the entire group (including the first message) from the queue using the same `MQGet` node.

Figure 5 .Summary of flow logic required to consume an MQ logical group


MQInput
Browse triggers flow on first message in a group
MQGet
Destructively get first/next message in group
Yes
More messages?
No
Done

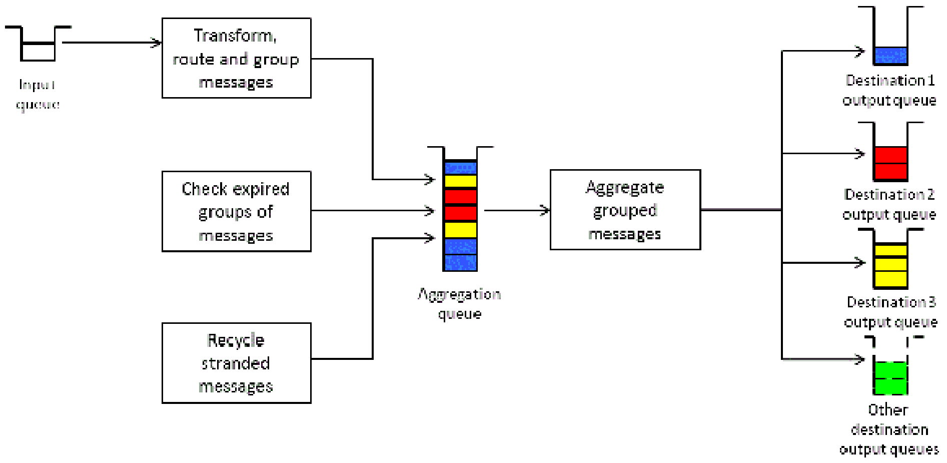
You may have noticed a loop in the above logic, and in general, you should not put loops in Message Broker graphical flows. The example below solves this problem by putting the loop in ESQL logic instead of in the graphical flow. It then invokes the `MQGet` node repeatedly using a `PROPAGATE` statement with `DELETE NONE` set.

The example application

The example, which you can download at the bottom of the article, demonstrates an implementation of custom message aggregation based on MQ-level message groups. The example application receives messages, transforms their content, determines their destination, groups the messages into aggregated messages by destination, and sends the aggregated messages to their destination. The maximum number of input messages to be aggregated into a single output message, and the time granted to build an aggregated message, are dependent on the destination.

Overview of example application

Figure 6. Example application


Input queue
Transform, route and group messages
Check expired groups of messages
Recycle stranded messages
Aggregation queue
Aggregate grouped messages
Destination 1 output queue
Destination 2 output queue
Destination 3 output queue
Other destination output queues

The `Transform, route, and group messages` functional part receives the messages to be processed from the input queue, transforms their content, determines their destination, assigns them to a group under construction, and puts them into the aggregation queue. The messages are grouped with other messages for the same destination.

The `Aggregate grouped messages` functional part waits for groups of messages in the aggregation queue to be complete. When one group is complete or overdue, it gets the messages of this group from the queue, aggregate them into a single output message, and puts the message into the proper destination queue.

Each destination defines a number of messages per aggregated message, and a time limit for gathering them. If throughput is not high enough to reach the specified number of messages in the alloted time, the `Check expired groups of messages` functional part identifies these overdue incomplete groups, and triggers their aggregation without waiting for additional messages.

In some situations, message can be stranded in the aggregation queue. In that case, the `Transform, route and group messages` and the `Aggregate grouped messages` parts have "forgotten" these messages and the under-construction groups to which they belong to. Therefore, there is no way for these groups to be completed and for the messages to be aggregated and forwarded. The `Recycle stranded messages` functional part deals with these stranded messages by retrieving and recycling them.

The example application contains three message flows:

TransformAndRouteMsg
: Main message flow. Receives and transforms the messages, determines their destination, groups them, and puts them into the aggregation queue. TransformAndRouteMsg also retrieves groups of messages that have been under construction for too long, and recycles stranded messages identified by the RecycleStrandedMsgs message flow.

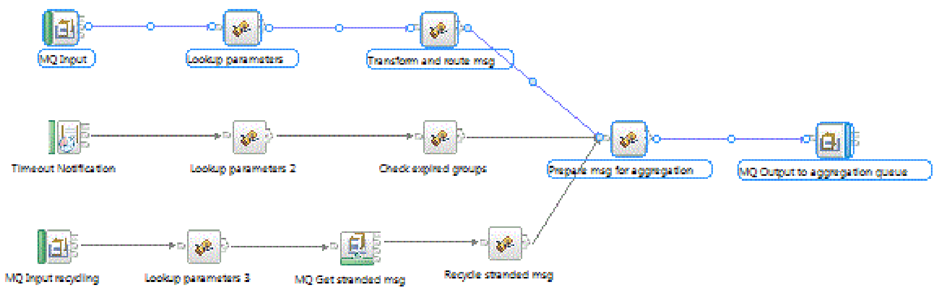
AggregateMsgs
: Aggregates messages belonging to complete groups in the aggregation queue and puts the resulting aggregated messages into the proper destination queues.

RecycleStrandedMsgs
: Retrieves messages stranded in the aggregation queue and triggers their recycling. Recycling is completed by the TransformAndRouteMsg message flow.

Main flow

The main flow of the application is part of the `TransformAndRouteMsg` message flow. It receives messages, transforms their content, determines their destination, assigns them to a group of messages under construction, and puts them into the aggregation queue.

Figure 7. Main flow (highlighted in TransformAndRouteMsg)



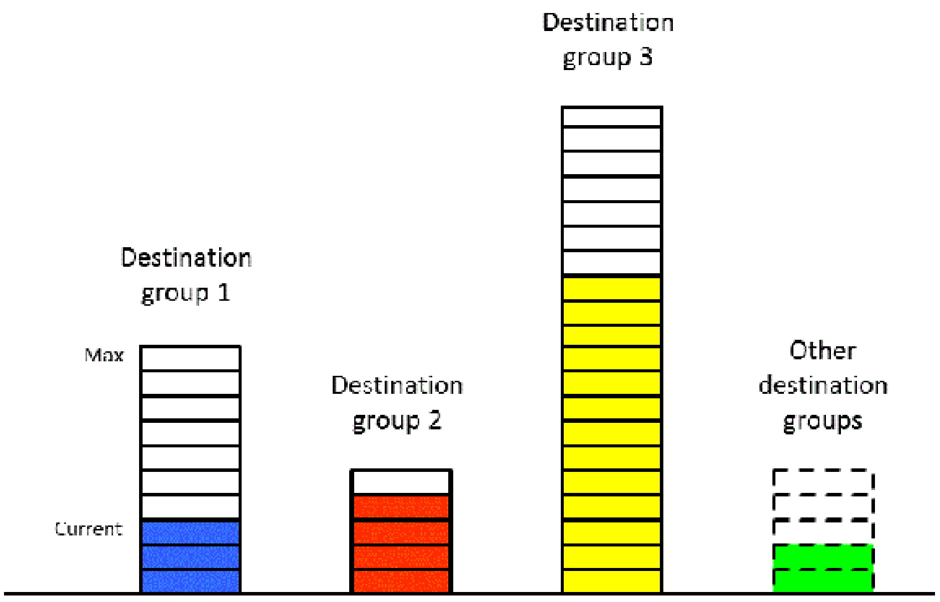
The main flow includes the following nodes:

- `MQ Input` node receives an XML message to be processed from an MQ queue.
- `Lookup parameters` node queries the configuration database to extract execution parameters dynamically. These parameters are not refreshed from the database every time, but saved in a memory cache for some time.
- `Transform and route` node transforms the message and determines its destination. Transformation has not been implemented because that is not the purpose of this article. For the routing, a destination name is extracted from the RFH2 header of the message, and searched for in the list of destination names obtained from the database. When a match is found, the target queue, the number of messages per aggregation group, and the time period to aggregate messages associated with the matching destination are selected.
- `Prepare msg for aggregation` node manages the groups of messages under construction and registers the current message in the group under construction associated with the destination of the message. The number of messages to be grouped together is defined by the destination. This node also gives the message a group id and a sequence number within the group, according to the following algorithm:
  1. Loop on the list of groups under construction, and search for the one associated with the destination of the current message. The list of groups under construction is stored in a shared variable of Type row, as shown in Figure 8 below.
  2. If no group with a matching destination is found, add a new group to the list and register the current message as the first one in this group.
  3. If a matching group is found and the group is neither empty nor complete, register the current message as a middle message of the group.

4. If a matching group is found and the group and has one free slot remaining, register the current message as the last message of the group.
5. If a matching group is found and the group is complete, clear the group and create a new group with a new unique id. Then register the current message as the first message of this new group.
6. Some fields are added to the current message: the group id and sequence number within the group are set in the MQMD header. If the message gets stranded and needs to be recycled, then additional fields are created in the RFH2 header.

- Eventually, the `MQ Output to aggregation queue` node puts the message into the aggregation queue.

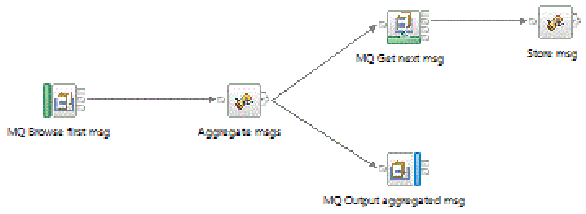
Figure 8. Information stored in the shared row used to build message groups



Aggregating messages

The `AggregateMsgs` message flow aggregates the messages using the groups defined by the `Prepare msg for aggregation` node of the main flow. When a group is completed, the node reads the messages belonging to the group from the aggregation queue (in sequence and without intermixing messages of other groups), then aggregates them into a single output message and puts the output message into the proper target queue.

Figure 9. Aggregating messages in AggregateMsgs message flow



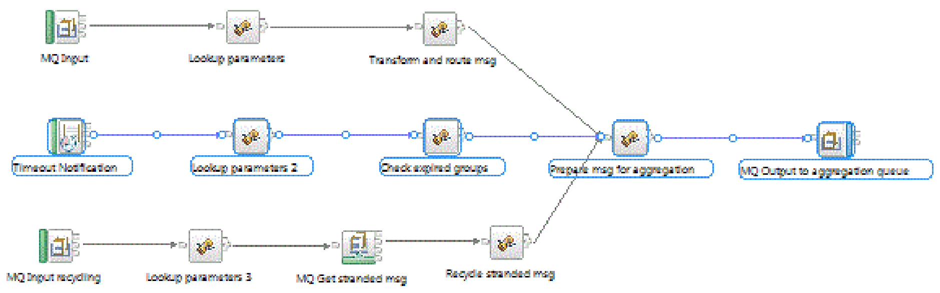
The `AggregateMsgs` message flow includes the following nodes:

- `MQ Browse first msg` node browses the first message of a group of messages in the aggregation queue when the group is complete.
- `Aggregate msgs` node executes a loop to read the messages of the group in sequence using the `MQ Get next msg` node, and aggregates the contents of these messages into a single output message.
- `MQ Get next msg` node reads the next message in the group from the aggregation queue.
- When all the messages of the group have been read, the `MQ Output aggregated msg` node puts the aggregated message into the target queue corresponding to the destination of the group.

Checking expired groups

There is time limit for grouping messages into an aggregated message, because it is not acceptable to keep messages indefinitely in groups under construction. This time limit is specified for each destination. The `Check expired groups` message flow deals with the time limit and is part of the `TransformAndRouteMsg` message flow.

Figure 10. Checking expired groups highlighted in TransformAndRouteMsg



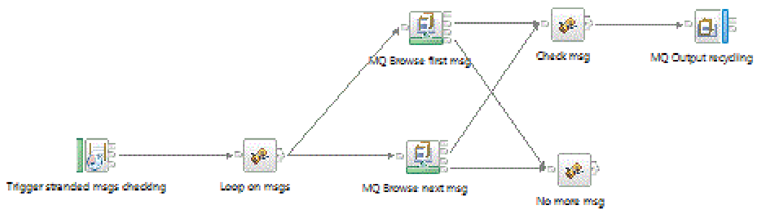
The `Check expired groups` message flow includes the following nodes:

- `TimeoutNotification` node generates a timeout notification every 15 seconds and triggers the checking of groups under construction.
- `Lookup parameters 2` node queries the configuration database to extract execution parameters dynamically. These parameters are not refreshed from the database every time, but saved in a memory cache for some time. This node shares the same ESQL code as the `Lookup parameters` node of the main flow.
- `Check expired groups` node loops on the groups under construction (stored in a shared variable of type Row) and checks that their expiration time is not exceeded. When an expired group is found, a timeout message is created and propagated to the next nodes in the flow. This node can generate several timeout messages per execution, depending on the number of expired groups found.
- `Prepare msg for aggregation` node receives the timeout message, retrieves the group under construction for which it was created, and registers the timeout message as the last one of the group, which closes the group and enables its processing by the `AggregateMsgs` flow.
- `MQ Output to aggregation queue` node puts the timeout message into the aggregation queue.

Recycling stranded messages

Sometimes, messages may remain in the aggregation queue while the group they belong to is incomplete and is no longer registered in the shared row containing groups under construction. These stranded messages may occur in particular when the application or execution group is stopped while groups of messages are still under construction. Special processing is needed to recycle these stranded messages so that they do not remain and accumulate in the aggregation queue., and this processing is handled by the `RecycleStrandedMsgs` and `TransforAndRouteMsg` message flows.

Figure 11. RecycleStrandedMsgs flow performs first stage of message recycling

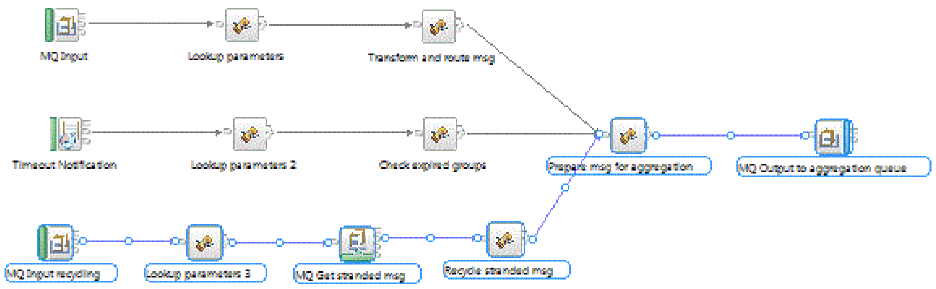


The `RecycleStrandedMsgs` message flow includes the following nodes:

- `Trigger stranded msgs checking` node generates a timeout notification every minute. This notification triggers the search for stranded messages.
- `Loop on msgs` node loops on the messages in the aggregation queue and requests the `MQ Browse first msg` or `MQ Browse next msg` nodes to browse them.
- `MQ Browse first msg` node browses the first message of the aggregation queue.
- `MQ Browse next msg` node browses the next message in the aggregation queue.
- `Check msg` node checks whether the message is stranded, based on the `stranded time` parameter, which specifies the time period after which the message is regarded as stranded, and is included in the RFH2 header of the message by the `Prepare msg for aggregation` node of the `TransformAndRouteMsg` message flow. If the message is stranded, a message including the message id of the message to be recycled (and empty content) is propagated to the `MQ Output recycling` node, to request the recycling of the message.
- `MQ Output recycling` puts the message requesting the recycling into the recycling queue. It is processed by the second stage of message recycling, in the `TransformAndRouteMsg` message flow.
- `No more msg` node registers that all the messages of the aggregation queue have been browsed and checked.

The second stage of message recycling is done in the `TransformAndRouteMsgs` message flow.

Figure 12. TransformAndRouteMsgs flow performs second stage of message recycling



`TransformAndRouteMsgs` message flow includes the following nodes:

- `MQ Input recycling` node receives recycling request message from the recycling queue, which was put there by the `MQ Output recycling` node of the `RecycleStrandedMsgs` flow. This message contains the message id of the message to be recycled.
- `Lookup parameters 3` node queries the configuration database to dynamically extract execution parameters . These parameters are not refreshed from the database every time, but are saved in a memory cache for a period of time. This node shares the same ESQL code as the `Lookup parameters` node in the main flow.
- `MQ Get stranded msg` node reads the stranded message from the aggregation queue, using the message id provided by the recycling request message.
- `Recycle stranded msg` node copies data needed to group the message from the RFH2 header to the LocalEnvironment tree.
- `Prepare msg for aggregation` node registers the recycled message in the current group under construction associated with the destination of the message. It also assigns a new group id and a new sequence number in the group to the message.
- Eventually, the `MQ Output to aggregation queue` node puts the recycled message into the aggregation queue.

Running the example application

This section shows you how to install and run the example application with Message Broker V8 on Microsoft® Windows® with DB2 as the database. The RFHUTIL tool is used to send input messages to the application and receive output messages from it.

Installing the example application

1. Download the zip file at the bottom of the article and extract it into a directory.
2. In order to create the DB2 configuration database used by the application to query its execution parameters, open a DB2 command window and execute the following commands:
   ```
   db2 create database CUSTAGGR
   db2 -tvf <dir>\CreateTable.ddl
   ```
   where <dir> is the directory where you extracted the zip file.
3. Create an ODBC definition named `CUSTAGGR` for to the `CUSTAGGR` database: Select **Administrative Tools => Data sources (ODBC)**. For details about this operation, see Connecting to a database from Windows systems in the Message Broker information center.

4. Specify the user ID and password for the broker to connect to the `CUSTAGGR` database: Open a Message Broker command window and execute the commands:
   ```
   mqsisetdbparms <broker> -n CUSTAGGR -u <userid> -p <password>
   mqsireload <broker> -e <execution_group>
   ```
   where <broker>, <execution_group>, <userid>, and <password> must be replaced with the correct values for your system.
5. Add queues with the following names to the MQ queue manager associated with the broker that will run the example application: `CAMIn, CMAOut.FR, CMAOut.UK, CMAOut.SP, CMAOut.OT, CMAAggr and CMARecycl`.
6. Import the `CustomMessageAggregation.zip` file as a Project Interchange file into a Message Broker Toolkit V8 workspace. Once the `CustomMessageAggregation` application is visible in the Broker Development view of the Toolkit, open the `BARs` folder, find the `CustomMessageAggregation.bar` file, and deploy it to an active execution group of your broker.

Scenario 1. Plain message aggregation

1. Start the RFHUTIL tool and open the `Test-FR.xml` file included in the downloadable zip file. Here is the content of this file, as shown on the Data page:
   ```
   <?xml version="1.0" encoding="utf-8"?>
   <Test>Salut mon pote</Test>
   ```
2. Go to the RFH page and select **Include RFH V2 Header** and **V2 Folders => user**. Go the Usr page and enter the following text into the Usr folder contents area: `destination=France`.
3. Return to the main page, specify the queue manager name and `CMAIn` for the queue, and then write the message 10 times into the `CMAIn` queue in less than 30 seconds.
4. Change the queue name to `CMAOut.FR` and read this queue. You will get a message with the following contents, which is the aggregation of the 10 input messages:
   ```
   <?xml version="1.0" encoding="utf-8"?>
   <Aggregation>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
      <Test>Salut mon pote</Test>
   </Aggregation>
   ```
   In this case, the aggregation process was straightforward: the messages where added to the `CMAAggr` queue until the maximum number of messages for the group (10 for the destination `France`) was reached. Then the messages were read, aggregated, and written to the target queue.

Scenario 2. Incomplete group completed by timeout

1. Open the `Test-UK.xml` file in RFHUTIL. Here are the contents of this file:
   ```
   <?xml version="1.0" encoding="utf-8"?>
   <Test>Hi buddy</Test>
   ```

2. On the RFH page, select **Include RFH V2 Header** and **V2 Folders => user**. On the Usr page, set the Usr property as follows: `destination=UK`.
3. Put this message three times into the `CMAIn` queue.
4. Less than one minute later, a message should arrive in the `CMAOut.UK` queue, with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<Aggregation>
   <Test>Hi buddy</Test>
   <Test>Hi buddy</Test>
   <Test>Hi buddy</Test>
</Aggregation>
```

In this case, the three messages where added to the `CMAAggr` queue, but were not enough to make the group complete (five messages are needed for the destination `UK`). Fifteen seconds after the first message was put, the group got expired and was identified so by the next run of the expired group checking. This generated a timeout message which caused the 3 messages to be aggregated and written to the `CMAOut.UK` queue.

Scenario 3. Stranded messages recycled

1. Open the `Test-SP.xml` file in RFHUTIL. Here are the contents of this file:

```
<?xml version="1.0" encoding="utf-8"?>
<Test>Hola amigo</Test>
```

2. On the RFH page, select **Include RFH V2 Header** and **V2 Folders => user**. On the Usr page, set the Usr property as follows: `destination=Spain`.
3. Put this message 12 times into the `CMAIn` queue in less than one minute, and then immediately stop and restart the `CustomMessageAggregation` application using Message Broker Toolkit.
4. After about four minutes, a message should arrive in the `CMAOut.SP` queue,with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<Aggregation>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
   <Test>Hola amigo</Test>
</Aggregation>
```

In this case, the 12 messages were added to the `CMAAggr` queue, but were not enough to make the group complete, because 20 messages are needed for the destination `Spain`. Moreover, when the `CustomMessageAggregation` application was stopped and restarted, it "forgot" these messages and the incomplete group that they belong to. Therefore, the expiration checking mechanism, which was able to close the group and trigger the aggregation in the previous scenario, did not work and the messages got stranded. The recycling mechanism then retrieved these stranded messages and recycled them.

When the `destination` property is not defined in the RFH2 header of the input message, or when it is defined but its value does not exist in the `CUSTAGGR` database, the following defaults are used:

- Default output queue: CMAOut.OT
- Default aggregation number of messages: 10
- Default aggregation delay: 60 seconds

Conclusion

The principles in the scenario in this article were recently used in a customer application. They can be very effective when implementing message aggregation in situations where you need additional flexibility beyond what is provided by the built-in Message Broker aggregation features.

Downloadable resources

Description	Name	Size
Code sample	ExampleApplication.zip	30 KB

Related topics

- **WebSphere Message Broker resources**
  - WebSphere Message Broker product page
    Product descriptions, product news, training information, support information, and more.
  - Download free trial version of WebSphere Message Broker
    WebSphere Message Broker is an ESB built for universal connectivity and transformation in heterogeneous IT environments. It distributes information and data generated by business events in real time to people, applications, and devices throughout your extended enterprise and beyond.
  - WebSphere Message Broker documentation library
    WebSphere Message Broker specifications and manuals.
  - WebSphere Message Broker forum
    Get answers to technical questions and share your expertise with other WebSphere Message Broker users.
  - WebSphere Message Broker support page
    A searchable database of support problems and their solutions, plus downloads, fixes, and problem tracking.
  - IBM Training course: WebSphere Message Broker V8 Development
    This course from IBM Training shows you how to use the components of the WebSphere Message Broker development and runtime environments to develop and troubleshoot message flows that use ESQL, Java, and PHP to transform messages.
- **WebSphere resources**
  - developerWorks WebSphere
    Technical information and resources for developers who use WebSphere products. developerWorks WebSphere provides product downloads, how-to information, support resources, and a free technical library of more than 2000 technical articles, tutorials, best practices, IBM Redbooks, and online product manuals.
  - developerWorks WebSphere application integration developer resources
    How-to articles, downloads, tutorials, education, product info, and other resources to help you build WebSphere application integration and business integration solutions.
  - Most popular WebSphere trial downloads
    No-charge trial downloads for key WebSphere products.
  - WebSphere forums
    Product-specific forums where you can get answers to your technical questions and share your expertise with other WebSphere users.
  - WebSphere demos
    Download and watch these self-running demos, and learn how WebSphere products can provide business advantage for your company.
  - WebSphere-related articles on developerWorks
    Over 3000 edited and categorized articles on WebSphere and related technologies by top practitioners and consultants inside and outside IBM. Search for what you need.
  - developerWorks WebSphere weekly newsletter
    The developerWorks newsletter gives you the latest articles and information only on those topics that interest you. In addition to WebSphere, you can select from Java, Linux,

    Open source, Rational, SOA, Web services, and other topics. Subscribe now and design your custom mailing.
  - WebSphere-related books from IBM Press
    Convenient online ordering through Barnes & Noble.
  - WebSphere-related events
    Conferences, trade shows, Webcasts, and other events around the world of interest to WebSphere developers.
- **developerWorks resources**
  - Trial downloads for IBM software products
    No-charge trial downloads for selected IBM® DB2®, Lotus®, Rational®, Tivoli®, and WebSphere® products.
  - developerWorks business process management developer resources
    BPM how-to articles, downloads, tutorials, education, product info, and other resources to help you model, assemble, deploy, and manage business processes.
  - developerWorks blogs
    Join a conversation with developerWorks users and authors, and IBM editors and developers.
  - developerWorks tech briefings
    Free technical sessions by IBM experts to accelerate your learning curve and help you succeed in your most challenging software projects. Sessions range from one-hour virtual briefings to half-day and full-day live sessions in cities worldwide.
  - developerWorks podcasts
    Listen to interesting and offbeat interviews and discussions with software innovators.
  - developerWorks on Twitter
    Check out recent Twitter messages and URLs.
  - IBM Education Assistant
    A collection of multimedia educational modules that will help you better understand IBM software products and use them more effectively to meet your business requirements.