Ultra Messaging (Version 6.12)

Guide to Queuing

Contents

Chapter 1

Introduction

This document describes the queuing functionality of the Ultra Messaging UMQ product.

Attention

> **See the Documentation Introduction for important information on copyright, patents, information resources (including Knowledge Base, and How To articles), Marketplace, Support, and other information about Informatica and its products.**

1.1 UMQ Overview

Ultra Messaging Queuing Edition (UMQ) is an extension of the functionality of UMS and UMP that provides message queuing functionality. Applications access UMQ features through the Ultra Messaging Application Programming Interface (API).

UMQ sources can send messages to a queue and receivers can retrieve messages from a queue asynchronously. UMQ includes the following functionality:

- Apache ActiveMQ™ broker
- High availability redundant broker configuration
- Queuing with persistence
- Once and only once delivery
- JMS (Java Message Service) support

UMQ also includes Ultra Load Balancing (ULB), for low-latency peer-to-peer load balancing without a broker.

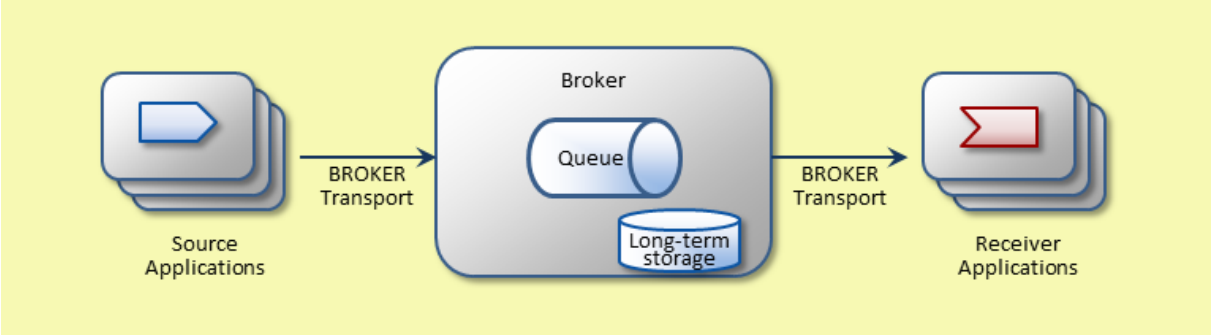
1.2 UMQ Architecture

An Ultra Messaging queuing system on a single node consists of client applications and a broker. An Ultra Messaging queuing system configured for high availability consists of client applications and three brokers.

1.2.1 Single Node

A single-node queuing system consists of client applications and a broker.

The following image shows the components of a single-node queuing system:



A single-node queuing system contains the following components:

Source Applications

Applications that you write, which contain a brokered context and one or more sources that send messages to broker queues. A source application can also be a JMS application.

BROKER Transport

The type of transport Ultra Messaging uses to connect client applications to UMQ brokers.

Broker

An Apache ActiveMQ broker. A broker contains one or many queues.

Queue

The message destination that holds messages until the messages are delivered to a client receiver application.

Long-Term Storage

The broker uses a message database for message persistence.

Receiver Applications

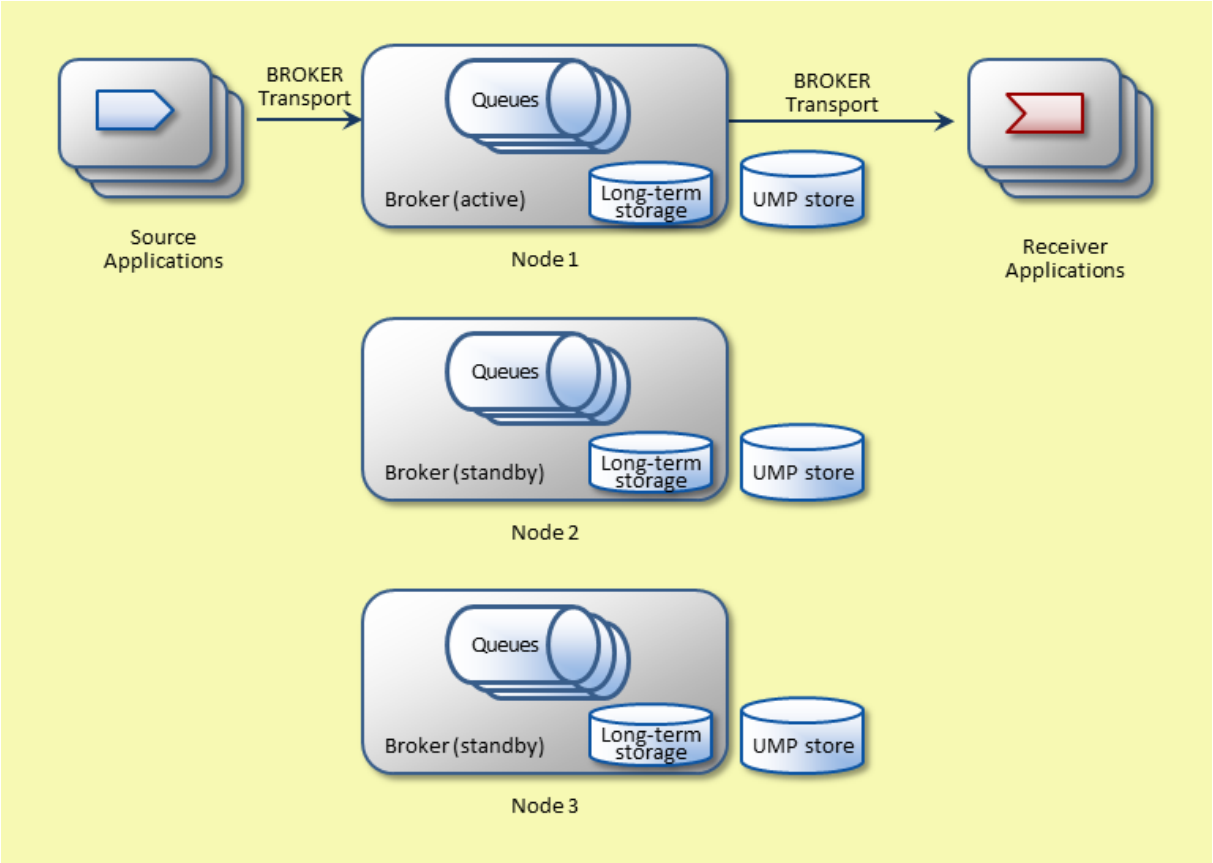
Applications that you write, which contain a brokered context and one or more receivers that receive messages from broker queues. A receiver application can also be a JMS application.

1.2.2 Introduction to High Availability

A high availability queuing system consists of client applications, three nodes, each with a broker, and three UMP stores. The UMP stores enable broker replication and failover functionality.

Client source and receiver applications connect to the active node only. The standby nodes receive a stream of replication updates from the active node, but the standby nodes do not generate any traffic except heartbeats and acknowledgments to the UMP stores.

The following image shows the components of a high-availability queuing system:



A high availability queuing system contains the following components:

Source Applications

Applications that you write, which contain a brokered context and one or more sources that send messages to broker queues. A source application can also be a JMS application.

BROKER Transport

The type of transport Ultra Messaging uses to connect client applications to UMQ brokers.

Node

A node consists of a broker and a UMP store.

Broker

An Apache ActiveMQ broker. A broker contains one or more queues. In a high-availability system, one broker functions as the active broker, and the other two brokers function as standby brokers.

Queue

The message destination that holds messages until the messages are delivered to a client receiver application.

Long-Term Storage

Each broker uses a message database for message persistence.

UMP Store

The UMP store holds state and message data for the broker. All UMP stores In a high-availability system hold the same data. If the active broker fails and a standby broker becomes the active broker, the UMP store of the new active broker contains the data necessary for it to continue storing and delivering messages.

UMP stores in a high-availability system hold the same data. As a group, the UMP stores ensure delivery of state-change and message-data events to all standby brokers. The active broker waits to process events and messages until a quorum of the UMP stores have acknowledged them to the active broker. The standby brokers rely on the stores for recovery of any missed events due to packet loss or a failure.

Receiver Applications

Applications that you write, which contain a brokered context and one or more receivers that receive messages from broker queues. A receiver application can also be a JMS application.

1.2.3 Client Applications

Client applications for queuing systems contain at least one brokered context. Client applications can contain Ultra Messaging sources or receivers, or both. Client applications can also include both standard contexts and brokered contexts.

1.3 Security and Authentication

Ultra Messaging message security depends on enterprise network security. Informatica does not support security and authentication features for ActiveMQ.

1.4 JMS

You can use a UMQ broker as a JMS provider for your JMS applications. You can write JMS message producer and message consumer applications that connect to and use a UMQ broker as a JMS provider.

You can write Ultra Messaging and JMS applications that interoperate in the following ways:

- Use an Ultra Messaging source application to send messages to queues in a UMQ broker, and use a JMS message consumer application to receive messages from the broker.
- Use a JMS message producer application to send messages to queues in a UMQ broker, and use an Ultra Messaging receiver application to receive messages from the broker.
- Use a JMS message producer application to send messages to topics or queues in a UMQ broker, and use a JMS message consumer application to receive messages from the broker.

You cannot use XA transactions, J2EE containers, or JCA resource adapters with Ultra Messaging.

Chapter 2

Getting Started

To install Ultra Messaging and Ultra Messaging options, download the Ultra Messaging installation files for your specific operating system and platform.

Ultra Messaging Queuing Edition installation files include the following files:

Core installation script

The core installation program or script is a self-extracting executable file you run to extract Ultra Messaging files to a host system. Use the core installation script that matches your operating system, version, and chip set combination.

Documentation

The compressed documentation files each contain the complete documentation set.

Option packages

Option packages include support for Java and .NET.

In addition to the installation files, you also receive the following files by email:

Software Download Instructions

Description of general procedure for downloading Informatica software.

License

You must supply the license to run Ultra Messaging applications.

2.1 Requirements

Ultra Messaging runs on a variety of current operating systems and chip sets.

2.1.1 Minimum System Requirements

For Ultra Messaging, there are no strict requirements for hardware computing capacity and storage capacity. Such requirements depend on client applications and on configuration of clients and brokers.

2.1.2 Software Requirements

UMQ brokers require Java SE Development Kit 6 or later. Informatica recommends Java SE Development Kit 7.

UMQ on UNIX requires an installation of a libstdc++ Standard C++ Library.

2.2 Single Node Installation

To install UMQ on a host for a single-node deployment, perform the following steps. All examples are Linux examples.

1. Consult your Informatica sales engineer for the download procedure and appropriate files for your specific operating system and platform. Informatica sends you the Ultra Messaging license file by email.
2. Download the installation files to the installation directory and run the installation script.
3. Locate the documentation .zip package and unzip it into the installation directory. For example:

   ```
   unzip UMQ_6.12_doc.zip -d <InstallDir>/
   ```

4. Copy the license file into a location that is accessible to the Ultra Messaging library.
5. Set the Ultra Messaging license environment variable to the full path name of the license file. For example:

   ```
   export LBM_LICENSE_FILENAME=/path/filename.lic
   ```

6. Add the library to the library path. For example, use the following line:

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<InstallDir>/UMQ_6.12/<platform>/lib
   ```

2.3 Starting a Single-Node Queuing System

After you have installed Ultra Messaging, run the daemons and sample applications to verify successful installation.

1. Start the broker with the following command:

   ```
   <InstallDir>/UMQ_6.12/<platform>/broker/apache-activemq-5.10.0/bin/activemq
       start
   ```

2. Start the sample receiving application with the following commands:

   ```
   <InstallDir>/UMQ_6.12/<platform>/bin/umqrcv -B <brokerIP:port> -r 10 QueueDestA
   ```

3. In a second command window, start the sample sending application with the following command:

   ```
   <InstallDir>/UMQ_6.12/<platform>/bin/umqsrc -B <brokerIP:port> -M 10 QueueDestA
   ```

   In the sending application window, you see output similar to the following example display:

```
0 msgs/sec -> 1 msgs/ivl, 0 msec ivl
Using flight size of 1000
Not using UME Late Join.
Using UMQ Message Stability Notification. Will not display events. Delaying
    for 1 second
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0][Inst: 10.29.3.49/
    5672] [0] ctx registration. ID 92da70611c65ece4 Flags 0
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0] ctx registration
    complete. ID 92da70611c65ece4 Flags 0
UMQ "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0] src registration
    complete. Flags 0
Sending 10 messages of size 25 bytes to topic [QueueDestA]
Sent 10 messages of size 25 bytes in 1.003 seconds.
1.003 secs. 0.009965 Kmsgs/sec. 1.993 Kbps
[BROKER], sent 10/1660, app sent 10 stable 0 inflight 10 Lingering for 5
    seconds...
Deleting source
Deleting context
```

In the receiving application window, you see messages being received, as in the following example display:

```
Request port binding disabled, no immediate messaging target.
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0][Inst: 10.29.3.49/
    5672] [0] ctx registration. ID a1387d282ba177d2 Flags 0
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0] ctx registration
    complete. ID a1387d282ba177d2 Flags 0
[QueueDestA][(Broker: Default IP: 10.29.3.49 PORT: 5672 )] UMQ "(Broker:
    Default IP: 10.29.3.49 PORT: 5672 )"[0] registration complete. AssignID 0.
    Flags 0
1.001 secs. 0
1.001 secs. 0
1.001 secs. 0
1.001 secs. 0
1.001 secs. 0
1.001 secs. 0.009991 Kmsgs/sec.
Quitting.... received 10 messages (0 reassigned, 0 resubmitted)
```

2.4 Shutting Down a Single-Node Broker

Shut down the broker with the following command:

```
<InstallDir>/UMQ_6.12/<platform>/broker/apache-activemq-5.10.0/bin/activemq stop
Kmsgs/sec. 0 Kmsgs/sec. 0 Kmsgs/sec. 0 Kmsgs/sec. 0 Kmsgs/sec. 0
Kbps Kbps Kbps Kbps Kbps 1.998 Kbps
```

2.5 High Availability Installation

A high-availability queuing system requires an Ultra Messaging installation on three hosts. To install UMQ on hosts for a high-availability queuing system, perform the following steps. All examples are Linux examples.

Note

Each node has a unique configuration file, but these configuration files are each in the same path- named directory. Therefore, the three installations must not be on the same NFS mounted disk.

1. Consult your Informatica sales engineer for the download procedure and appropriate files for your specific operating system and platform. Informatica sends you the Ultra Messaging license file by email.
2. Download the installation files to the installation directory of the first broker host and run the installation script.
3. Locate the documentation .zip package and unzip it into the installation directory. For example:

   ```
   unzip UMQ_6.12_doc.zip -d <InstallDir>/
   ```

4. Copy the license file into a location that is accessible to the Ultra Messaging library.
5. Set the Ultra Messaging license environment variable to the full path name of the license file. For example:

   ```
   export LBM_LICENSE_FILENAME=/path/filename.lic
   ```

6. Add the library to the library path. For example, use the following line:

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<InstallDir>/UMQ_6.12/<platform>/lib
   ```

7. A high availability broker node contains a working configuration directory and a default configuration directory for each unique node in a multiple node arrangement.

   Create the store repository directories. Then, depending on which host you are configuring, copy the default broker1, broker2, or broker3 configuration file to the working configuration directory with the following example commands:

   ```
   mkdir <InstallDir>/UMQ_6.12/<platform>/broker/store/
   mkdir <InstallDir>/UMQ_6.12/<platform>/broker/store/cache
   mkdir <InstallDir>/UMQ_6.12/<platform>/broker/store/state
   cp <InstallDir>/UMQ_6.12/<platform>/broker/deployments/
       three_replicating_brokers/broker1/um.xml <InstallDir>/UMQ_6.12/<platform>/
       broker
   ```

8. Edit the um.xml configuration file that is in <InstallDir>/UMQ_6.12/<platform>/broker/ to set the resolver IP addresses and ports for all three nodes, as shown in the following example:

   ```
   <option name="resolver_unicast_daemon" default-value="10.33.200.201:21212"/>
   <option name="resolver_unicast_daemon" default-value="10.33.200.202:21212"/>
   <option name="resolver_unicast_daemon" default-value="10.33.200.203:21212"/>
   ```

   Also, set the local broker host interface addresses to the IP address of a network interface on the current broker host that has connectivity to the other broker hosts, and to the servers where the clients run, as shown in the following example:

   ```
   <option name="resolver_multicast_interface" default-value="10.33.200.201"/>
   <option name="resolver_unicast_interface" default-value="10.33.200.201"/>
   <option name="request_tcp_interface" default-value="10.33.200.201"/>
   <option name="response_tcp_interface" default-value="10.33.200.201 "/>
   ```

   Note

   Do not change any values in the broker um.xml configuration file other than interfaces and addresses shown in this step.

9. Depending on which host you are configuring, copy the default broker1, broker2, or broker3 stored.xml configuration file to the working configuration directory with the following broker1 example command:

   ```
   cp <InstallDir>/UMQ_6.12/<platform>/broker/deployments/
       three_replicating_brokers/broker1/store/stored.xml <InstallDir>/UMQ_6.12/
       <platform>/broker/store
   ```

10. Depending on which host you are configuring, copy the default broker1, broker2, or broker3 ActiveMQ configuration file to the working configuration directory with the following broker1 example command:

```
cp <InstallDir>/UMQ_6.12/<platform>/broker/deployments/
    three_replicating_brokers/broker1/apache-activemq-5.10.0/conf/activemq.xml
    <InstallDir>/UMQ_6.12/<platform>/broker/apache-activemq-5.10.0/conf
```

11. Repeat steps 1-10 for the second broker host, and again for the third broker host, making sure to copy the configuration files from ...three_replication_brokers/broker2/... or ...three_replication_brokers/broker3/... as needed.

2.6 Starting a High Availability Queuing System

After you have installed Ultra Messaging, run the daemons and sample applications to verify successful installation.

1. On the first host, start the lbmrd resolver daemon with the following example command:

```
<InstallDir>/UMQ_6.12/<platform>/bin/lbmrd -i 10.33.200.201 -p 21212 &
```

2. Start the umestored daemon with the following example commands:

```
cd <InstallDir>/UMQ_6.12/<platform>/broker/store
<InstallDir>/UMQ_6.12/<platform>/bin/umestored stored.xml &
```

3. Start the broker with the following example commands:

```
cd <InstallDir>/UMQ_6.12/<platform>/broker/apache-activemq-5.10.0
<InstallDir>/UMQ_6.12/<platform>/broker/apache-activemq-5.10.0/bin/activemq
    start
```

4. Repeat steps 1-3 on the second and third host.

5. Start the sample receiving application with the following commands:

```
<InstallDir>/UMQ_6.12/<platform>/bin/umqrcv -B
    <broker1IP:port1,broker2IP:port2,broker3IP:port3> -r 10 QueueDestA
```

6. In a second command window, start the sample sending application with the following command:

```
<InstallDir>/UMQ_6.12/<platform>/bin/umqsrc -B
    <broker1IP:port1,broker2IP:port2,broker3IP:port3> -M 10 QueueDestA
```

   In the sending application window, you see output similar to the following example display:

```
0 msgs/sec -> 1 msgs/ivl, 0 msec ivl
Using flight size of 1000
Not using UME Late Join.
Using UMQ Message Stability Notification. Will not display events. Delaying
    for 1 second
LOG Level 4: Core-8901-03: BROKER: Disconnect detected while establishing
    broker connection (10.29.3.85:5672).
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0][Inst: 10.29.3.49/
    5672] [0] ctx registration. ID f544ce868d92a087 Flags 0
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0] ctx registration
    complete. ID f544ce868d92a087 Flags 0
UMQ "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0] src registration
    complete. Flags 0
Sending 10 messages of size 25 bytes to topic [QueueTstA] Sent 10 messages of
    size 25 bytes in 1.003 seconds. 1.003 secs. 0.009972 Kmsgs/sec. 1.994 Kbps
[BROKER], sent 10/1660, app sent 10 stable 0 inflight 10 Lingering for 5
    seconds...
Deleting source Deleting context
```

In the receiving application window, you see messages being received, as in the following example display:

```
Request port binding disabled, no immediate messaging target.
LOG Level 4: Core-8901-03: BROKER: Disconnect detected while establishing
    broker connection (10.29.3.85:5672).
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0][Inst: 10.29.3.49/
    5672] [0] ctx registration. ID a8c3c09a7ccb1476 Flags 0
UMQ queue "(Broker: Default IP: 10.29.3.49 PORT: 5672 )"[0] ctx registration
    complete. ID a8c3c09a7ccb1476 Flags 0
[QueueTstA][(Broker: Default IP: 10.29.3.49 PORT: 5672 )] UMQ "(Broker:
    Default IP: 10.29.3.49 PORT: 5672 )"[0] registration complete. AssignID 0.
    Flags 0
1.001 secs. 0      Kmsgs/sec.  0          Kbps
1.001 secs. 0      Kmsgs/sec.  0          Kbps
1.001 secs. 0      Kmsgs/sec.  0          Kbps
1.001 secs. 0      Kmsgs/sec.  0          Kbps
1.001 secs. 0      Kmsgs/sec.  0          Kbps
1.001 secs. 0.007993 Kmsgs/sec.  1.599 Kbps
1.001 secs. 0.001998 Kmsgs/sec.  0.3996 Kbps
Quitting.... received 10 messages (0 reassigned, 0 resubmitted)
```

2.7 Shutting Down High Availability Nodes

To shut down a high availability queuing system node, perform the following steps:

1. Shut down the broker with the following command:

   ```
   <InstallDir>/UMQ_6.12/<platform>/broker/apache-activemq-5.10.0/bin/activemq
       stop
   ```

2. Shut down the umestored process.
3. Shut down the lbmrd process.
4. Repeat steps 1-3 for the remaining nodes.

2.8 Queuing Upgrade Considerations

When UMQ 6.8 was released, it added some capabilities and deprecated others, when compared with pre-6.↩ 8 UMQ. You might need to change how you set configuration options or change function calls in your application code.

If the application does not use UMQ-specific features other than Ultra Load Balancing (ULB), ignore this section. This section is a supplement to upgrade guidelines in the Release Notes and does not repeat the guidelines therein.

2.8.1 Configuration Options

When you upgrade client applications from versions prior to 6.8, you must change the following configuration settings:

- Contexts that contain queue sources or receivers must specify a list of brokers with the broker option.
- Sources in a brokered context must set source option transport to BROKER.

2.8.2 Application Code

You might need to make changes to client application source code.

You do not need to change any API calls unless the functions are deprecated. However, you must recompile client applications from versions prior to 6.8.

UMQ client applications from versions prior to 6.8 might use the same context for both queue sources or receivers and topic sources or receivers. In such cases, you must create separate contexts for queue applications and non-queue applications.

For more information, see Deprecated and Unavailable Features.

Chapter 3

Concepts and Features

An Ultra Messaging queuing system uses a broker to manage the transfer of messages from a source to a receiver. When you build an Ultra Messaging queuing system, identify the broker to use and write a client application that defines the behavior of the source and the receiver.

You can configure the broker to use the Ultra Messaging persistence adapter. When you use the Ultra Messaging persistence adapter, the broker uses Ultra Messaging Persistence (UMP) functionality to manage the event stream and save messages to long-term storage.

To implement high availability, you can install and configure brokers on multiple nodes. A high availability queuing system includes an active broker and multiple standby brokers. Each broker maintains a copy of the messages and event stream.

Ultra Messaging generates events that you can listen for to determine the state of the source, receiver, and messages. You can write a callback function to check for and handle events according to the requirements of the queuing system.

You can use an index to group messages and send them to a specific receiver for processing. Use indexed queuing to ensure that all related messages are processed in the correct order.

You can configure a composite destination in the broker to distribute messages from one source to different receivers based on specific selection criteria. You can also use a composite destination to send messages to one receiver and send copies of the messages to another receiver.

3.1 Brokered Context

The brokered context establishes the environment for an Ultra Messaging queuing system.

When you build an Ultra Messaging queuing system, you must create a context and define the broker for the context. You can have multiple brokered contexts in a queuing system.

You can define the broker for the context in the client configuration file or in the application code. When you define the broker for a context, you must specify the IP address of the machine that hosts the broker and the Advanced Message Queuing Protocol (AMQP) listening port.

To define the broker for a brokered context in a configuration file, the option **broker (context)** as follows:

```
context broker broker_ip_address:port
```

For more information about the Ultra Messaging context, see **Context Object**.

3.2 Brokered Source

In an Ultra Messaging queuing system, the source is the component that sends messages to a broker.

In a client application, you must create a source object that connects and sends messages to the broker. You can include metadata with a message through message properties. The client application can listen for events to determine the state of the source, message, or broker.

When you create a source, you must specify the name of a message destination. The broker creates a queue destination with the specified name. If you create multiple sources, specify a different message destination for each source. A source sends messages to the message destination associated with it.

You can configure the source in the configuration file or in the application code. When you configure the source for a brokered context, you must set the transport option to broker.

To set up the broker transport for a source in the client configuration file, include the following parameter:

```
source transport broker
```

3.2.1 Source Event Concepts

Ultra Messaging generates source events that you can listen for and respond to based on your requirements.

You can use source events to determine the state of the source and the messages sent by the source. For example, when you write a callback function to respond to source events, you can check for a registration complete event and start sending messages only after you receive the event. You can also check events to determine whether a message is stable or whether the memory allocated to the message is reclaimed.

Ultra Messaging provides other events that you can check to manage the message queuing process, including events such as a registration error and flight size change.

For more information about source events, see Source Events.

3.3 Brokered Receiver

A receiver is the component in an Ultra Messaging queuing system that consumes the messages that are sent by a source. The source sends messages to a broker and the broker assigns messages to the receiver. The receiver does not directly access messages from the source.

In a client application, you must create a receiver object that connects to and gets messages from a broker. The receiver connects to the broker defined in the context. When you create a receiver, you must specify the message destination in the broker from which to receive messages.

A receiver must be registered with a broker before the receiver can receive messages from the broker. When you create a receiver, you trigger a receiver registration. A receiver registration complete event is a receiver event that indicates that the receiver is registered and can receive messages from the broker.

When a receiver consumes a message, it sends an acknowledgment to the broker. If the broker does not get an acknowledgment from the receiver after it assigns a message, the broker could resend the message to the receiver. When the broker resends the messages, it sets the LBM_MSG_FLAG_RETRANSMIT flag to indicate that the message has been sent previously and could be a duplicate. In the client application, you must check for the LBM_MSG_FLAG_RETRANSMIT flag to determine if a message is a duplicate and then take action based on your requirements.

3.4 The Broker

The message broker manages the transfer of messages from the source to a receiver. The broker receives messages from the source and assigns them to a receiver based on the message queue configuration.

The Ultra Messaging message broker provides the following services:

- Connection services. The broker manages the connections between the source and the broker and between the broker and the receiver.
- Message delivery services. The broker assigns and delivers messages and generates events to provide information about the state of the broker.
- Persistence services. The broker writes messages and events to storage.
- High availability services. The broker defines a high availability process that uses Ultra Messaging functionality to save messages and replicate them to multiple brokers.

Note

The Ultra Messaging message broker is built on top of the Apache ActiveMQ message broker. The Ultra Messaging Queuing API connects to the Advanced Message Queuing Protocol (AMQP) transport connector in ActiveMQ. The Ultra Messaging broker uses the same default listening port number 5672 that ActiveMQ uses. The Ultra Messaging broker also uses the same default file name activemq.xml for the broker configuration file. You can change the port number and configuration file name for the Ultra Messaging broker.

3.4.1 Message Destination

The message destination is the location on the broker where a message is stored before the broker assigns the message to the receiver.

When you create a source, you must specify the name of the message destination to which the source sends messages. When you create a receiver, you must specify the message destination from which the receiver consumes messages.

Note

Ultra Messaging uses the term topic to refer to the message destination. However, the message destination created in an Ultra Messaging queuing system is a queue, not a topic.

3.4.2 Message Store

You can configure the Ultra Messaging broker to use the Ultra Messaging persistence adapter that replicates events generated by the active broker to the standby brokers. When you use the Ultra Messaging persistence adapter, each broker maintains an independent message store.

The Ultra Messaging broker uses Ultra Messaging Persistence (UMP) functionality to manage events generated within the Ultra Messaging queuing system. The active broker uses a UMP source to replicate events to the standby brokers. A standby broker uses a UMP receiver to receive events from the active broker. If a broker fails, it uses a UMP store daemon to recover events that it missed when it was down.

After a standby broker receives events from the active broker, the standby broker stores the messages locally in the default long-term storage.

3.5 Flight Size Concepts

When a source sends a message, the message is considered to be in-flight until the broker generates a message stable event. Flight size is the number of messages that are in-flight at any point in time.

You can set a maximum flight size for a source to control the message load on the broker. Set the flight size based on the requirements of the queuing system and the capacity of the network. In an Ultra Messaging queuing system, setting the flight size flight size is the only method available to limit the volume of messages from the source.

3.6 High Availability Concepts

You can install and configure brokers on multiple nodes to implement high availability in an Ultra Messaging queuing system.

When you configure the context for a multi-node queuing system, you must specify the number of brokers and the IP address and port number of each broker. A queuing system set up for high availability has one active broker and multiple standby brokers.

The source sends messages to the active broker and the active broker assigns messages to the receiver. If the active broker fails, Ultra Messaging elects one of the standby brokers as an active broker. When the failed broker restarts, it runs as a standby broker and synchronizes with other brokers to recover events that it missed while it was down.

To have a successful failover, each broker must have the same copy of the messages and event stream. If a standby broker is elected as the active broker, the standby broker must have the same data as the previous active broker to be in the correct state to start as an active broker.

The broker in each node stores a copy of the messages and event stream. The active broker replicates the event stream to the UMP store daemons and the standby brokers. The active broker does not send a message stable event back to the client or save a message to long term storage until the replicated event stream is stable in more than half of the UMP store daemons.

3.7 Indexed Queuing

Indexed queuing is a way to group messages and ensure that one receiver processes all messages in the group in the order that they are sent. The source uses an index to specify the messages in a group. The Ultra Messaging broker uses the ActiveMQ message group functionality to implement indexed queuing.

If a set of messages must be processed in order, it can be more efficient for one receiver to process all the messages. For example, if the messages represent bids in an auction, you might want to assign all bids for an item to one receiver to ensure that the bids are processed in the right order. Include the same index in all messages that relate to an item to ensure that all bids for the item go to one receiver.

To set up indexed queuing, define an index with a string value and add the index to the message as metadata. By default, the broker checks the index of a message before it assigns the message to a receiver. The broker designates a receiver to process messages with a specific index. When the source sends a message with the index, the broker assigns the message to the designated receiver. The broker assigns all messages with the same index to the same receiver.

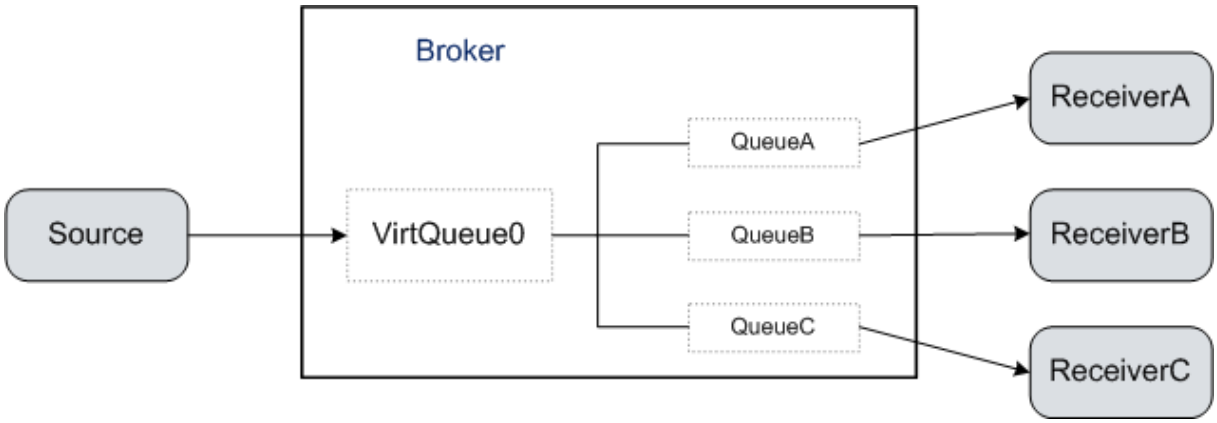
In the client application, extract the index to determine the message group and process the message based on your requirements.

3.8 Composite Destinations

A composite destination is a virtual message destination that represents a collection of physical message destinations.

You can create a source that sends messages to a virtual queue destination in the broker. In the broker configuration, you associate the virtual queue destination to multiple physical queue destinations. You can then create a receiver to receive messages from one of the physical queue destinations.

The following image shows an example of a broker configured with a composite destination:



In this example, the broker has a virtual queue and three physical queues. The source sends messages to the virtual queue. The broker forwards the message to the physical queues. The receivers get messages from the physical queues.

By default, if the broker is configured to use a composite destination, the source cannot send messages directly to one of the associated physical destinations. When you create a source, you specify the name of the virtual queue destination for the messages sent by the source.

Inversely, the receiver cannot consume messages from a virtual message destination. When you create a receiver, you must specify the name of a physical queue destination from which to receive messages sent by the source.

For example, to add auditing capabilities to the queueing system, you can use a composite destination to write all messages to a log. You can set up the source to send messages to a virtual queue named VirtQueue. In the broker, forward the messages in VirtQueue to physical message queues named ProcessQueue and LogQueue. Create a receiver to get messages from ProcessQueue and process the messages. Create another receiver to get messages from LogQueue and write the messages to a log file.

You can also use a composite destination to distribute messages from the source to different receivers based on specific selection criteria. Configure the broker to apply the selection criteria to each message in the virtual queue before it forwards the message to a physical queue.

3.8.1 Composite Destination Configuration

You configure the composite destination in the Ultra Messaging message broker. Use the broker configuration file to set up the logical and physical queues for the composite destination.

The following XML code shows an example of a configuration for a composite destination:

```
<destinationInterceptors>
    <virtualDestinationInterceptor>
        <virtualDestinations>
            <compositeQueue name="VirtQueue0">
                <forwardTo>
```

```
                    <queue physicalName="QueueA" />
                    <queue physicalName="QueueB" />
                    <queue physicalName="QueueC" />
                </forwardTo>
            </compositeQueue>
        </virtualDestinations>
    </virtualDestinationInterceptor>
</destinationInterceptors>
```

3.9 Message Lifetime

The message lifetime is the length of time within which the broker must assign a message to a receiver before the message expires. The message lifetime starts when the broker receives the message and expires after a length of time that you can configure. When the lifetime of a message expires, you can configure the broker to discard the message or send the message to the dead letter queue (DLQ).

You can set the message lifetime when you send a message or you can set the message lifetime as a configuration option. By default, if you do not set a message lifetime, a message does not expire and can be redelivered to a receiver indefinitely.

3.9.1 Dead Letter Queue

The dead letter queue (DLQ) refers to a message destination for messages that cannot be delivered to a receiver.

The Ultra Messaging broker contains a default DLQ message destination for all undeliverable messages. When the lifetime of a message expires or the maximum number of redelivery attempts is reached, the broker sends the message to the DLQ.

You can set up a separate DLQ for each message destination. In the broker configuration file, you can set up a dead letter strategy for a destination and create a DLQ associated with the destination. The broker sends the undeliverable messages to the DLQ associated with a message destination.

You can create a receiver to process the messages in the DLQ. Review the messages to look for common patterns and possible causes for delivery failure.

Chapter 4

Best Practices

When you use queuing, you should follow best practices to ensure optimal performance, reliability, and efficient use of resources.

When you write client applications, consider how you create and use contexts, how you time connection initiation, and the use of message persistence. When you set configuration options, consider whether to do this with configuration files or programmatically. Configuration options can help you tune applications for the appropriate balance between performance and reliability. When you deploy a queuing system, consider using a high availability deployment.

4.1 Application Development Best Practices

To ensure reliable operation of UMQ source and receiver applications, consider the following best practices.

4.1.1 Use the Appropriate Number of Contexts

Usually a client application needs only one context. However, a client application might need more than one context, as in the following examples:

- You cannot use the same context for UMQ sources and receivers and UMS or UMP sources and receivers. If your applications uses both types of source and receiver, create a separate context for each type.
- Because a brokered context can make only one connection to a broker, you might want to configure multiple sources or receivers with separate connections to a broker to increase throughput. To do this, create a separate context for each source or receiver.

Define a context and set default attributes with a configuration file. Then define additional contexts and set their attributes in the client application code.

4.1.2 If Performance Is Important, Consider Not Persisting Messages

By default, source applications flag messages to the broker as persisted messages. You can unset this flag to send the messages as non-persisted. Non-persisted messages do not provide as much reliability, but tend to reach their

destination faster because they do not need to wait for persistence operations to complete. Thus, the decision to persist messages or not can be a choice of performance versus reliability.

4.1.3 Wait for Registration

To avoid start-up errors, do not start sending messages immediately after source creation. Instead, wait for sources to register with the broker.

After you create the source, wait for the **LBM_SRC_EVENT_UMQ_REGISTRATION_COMPLETE_EX** event before sending messages.

4.1.4 Plan For Possible Connection Interruptions

A broker might stop or lose connection to a client application while an operation is in progress. Plan for this possibility when writing source or receiver applications.

If a broker stops and then restarts, a source application might not receive stability acknowledgements for in-flight messages. In this situation the source application must resend the message if needed, and possibly mark the message as previously sent.

If a broker or a receiver stops and then restarts, a receiver application might receive duplicate redelivered messages, and must handle these appropriately.

4.1.5 Java and C# Receivers: Call dispose()

For Java and C# applications, when you complete processing a message, call the dispose() method to reuse the LBMMessage object. This can significantly increase performance.

4.2 Configuration Best Practices

You can configure queuing application attributes by setting default values in a configuration file, then, if needed, setting values in the code to override these defaults.

4.2.1 Use Configuration Files

Ultra Messaging configuration files set attributes for contexts, sources, or receivers. As a client application launches, it reads these files. Within client application code you can set attribute values to override those set by the configuration file. For non-default attributes within objects, you must decide whether to set them in a configuration file or within the client application code.

Use a UM configuration file to set attributes with the following characteristics:

- settings that are generic or common to most objects

- settings that are likely to change, such as broker IP address and port number

When using configuration files to set attributes for client applications, you can use either plain text configuration files or XML configuration files. Become familiar with the advantages of both types before deciding which type to use. For complete information about how to use both types of configuration files, see **Configuration Overview**.

The plain text configuration file, when invoked, writes option values into Ultra Messaging current default attributes. Ultra Messaging then uses these values in the creation of all objects. The format is simple, and Ultra Messaging does not validate the file syntax.

XML configuration files override default attributes and have more flexibility than a plain text configuration file. A single XML file can contain options for multiple client applications. For a single application, you can configure multiple named contexts, event queues, and so on, with different values for the same options. Also, Ultra Messaging Manager (UMM) can manage XML configuration files.

4.2.2 Set Permanent Attributes Within the Code

Use function calls in the client application to set attributes that are not expected to change. Also use function calls in the client application to set attributes that are unique to the object.

4.2.3 Consider Nonblocking Sources

Decide if you want client application sources to be blocking or nonblocking.

Problems with the broker or transport, such as when flight size is exceeded, can cause sources to block, and to stop sending messages. You can configure client applications such that when Ultra Messaging instructs a source to block, the source does not block or send, but instead returns with an LBM_EWOULDBLOCK error.

4.3 Deployment Best Practices

When deploying a UMQ queuing, consider such things as whether to use a high-availability system and how to manage configuration files.

4.3.1 Use a Three-Node System for Greater Reliability

If persisted or queued data is critical, or if continuity of service is critical, use a three-broker high availability deployment. The high-availability configuration can protect against such problems as a host restart, hard- drive failure or entire system failure.

Note

A high availability UMQ queuing system is available in only a three-node configuration.

4.3.2 Keep Track of Configuration Files

Brokered client application configuration file management is more complex than for non-brokered client applications. Queuing systems might require some or all of the following configuration files:

- An Ultra Messaging configuration file for each client application.
- An ActiveMQ configuration file for each broker.
- An Ultra Messaging configuration file for each broker. The ActiveMQ configuration file references this Ultra Messaging configuration file.

Ensure that configuration files reside in a manageable location. Keep records and backups of the entire set of configuration files for a queuing system. Also consider using a configuration management tool such as Ultra Messaging Manager or a content management system.

4.4 Memory Usage

Many UM users need to modify ActiveMQ's default memory allocations.

4.4.1 JVM Size

The ActiveMQ broker defaults to having a minimum and maximum JVM size of 1GB. Many UM users will find that this size restriction is inadequate. This can be configured by overriding the ACTIVEMQ_OPTS_MEMORY environment variable. For example, to set it to 1GB minimum and 2GB maximum, set the following prior to invoking ActiveMQ's startup script:

```
export ACTIVEMQ_OPTS_MEMORY="-Xms1G -Xmx2G"
```

4.4.2 JVM Reserved Space

By default, ActiveMQ is allowed to use 70% of the JVM heap space, with the remaining reserved for headroom. You can modify the activemq.xml file's systemUsage element to modify that percentage. See below for advice on setting reserved space.

4.4.3 Message Paging

When a destination is initialized, ActiveMQ defaults to paging in 200 messages for that destination. If your messages are so large that paging in 200 messages from a destination will cause you to exceed the configured JVM limit, you should lower the number of messages to page in per destination. This can be done by adding the following attribute to any policy entries that have large messages in the activemq.xml file:

```
<policyEntry queue=">" persistJMSRedelivered="true" maxPageSize="20"></policyEntry>
```

The values you use will depend on your maximum message size. Using the default configured values, the JVM gets 1GB of memory and ActiveMQ is allowed to use 70% of that memory (700MB) with 300MB reserved for headroom. Knowing that ActiveMQ could page in 200 messages on startup, you need to make sure that your 200 messages will fit inside the 30% headroom. This is because when ActiveMQ begins paging in messages it does a one time memory check. If you were at 69% of the JVM limit, it will see that you have space and then page in 200 pages. If the 200 messages it pages in don't fit in the remaining headroom space, you will encounter memory errors.

For example, consider a case where you are sending 10MB messages. With the default page size of 200, you would need 2GB of available to page in all of those messages. This value is much greater than the 30% headroom available, so you could get memory errors.

This can be handled in two ways:

1. Reducing the page size to 20 messages will lower the memory requirement to 200MB (20 messages $*$ 10MB each), which will fit in the 30% headroom space of the 1GB JVM size.
2. Increasing the JVM size to 8GB will allow you to page in all 200 messages, because the 30% reserved space of an 8GB JVM is 2.4GB.

It would also be possible to change the systemUsage element to modify the 30% default for reserved space.

Chapter 5

Developing the Client Application

A queuing client application can have a combination of sources and receivers.

To write queuing client applications, use standard Ultra Messaging programming practices. A UMQ queuing context must be able to establish broker connections. If the client application contains non-brokered sources or receivers, these sources or receivers must use a separate, non-brokered context.

Writing UMQ source and receiver applications includes creating brokered contexts, creating sources, creating receivers, and setting attributes by configuration files or other methods.

Coding examples are in C language.

5.1 Before You Start

Before starting to develop client applications, decide on a language, verify that the Ultra Messaging libraries work with your development environment, and determine a general approach for setting configuration options.

Decide on the language to write sending and receiving client applications.

You can use C, Java, or for the .NET framework, C#. To use Java or .NET, you must install the optional Java or .NET Ultra Messaging APIs.

Decide how to set configuration options.

For options that are unique to the application and which are not expected to change, you might want to use attribute-setting functions or methods. To be able to select and set option values without recompiling, use configuration files.

Verify that the correct libraries are installed and accessible by the client applications.

You must install the following libraries:

- Ultra Messaging C API and included third-party APIs
- For Java or .NET, the appropriate Ultra Messaging Java or .NET API

Decide on locations for Ultra Messaging and broker configuration files

Environment variables or configuration file calls point to these locations.

5.2 Create the Context

A client application that sends or receives messages must have one or more context objects.

When you create a brokered context, set the IP address and port of the broker, and configure the attributes in a context attribute object. After you create the context, you can delete the context attribute object. The following code fragments show context coding concepts.

Create a context attribute object with the current default configuration.

```
if (lbm_context_attr_create(&cattr) == LBM_FAILURE) {
    fprintf(stderr, "lbm_context_attr_create: %s\n", lbm_errmsg());
    exit(1);
}
```

Set the configuration options that must be unique or permanent for the context. The following example sets configuration option broker to 10.29.3.190:5672.

```
if (lbm_context_attr_str_setopt(cattr, "broker", "10.29.3.190:5672") ==
    LBM_FAILURE) {
    fprintf(stderr, "lbm_context_attr_str_setopt:broker: %s\n", lbm_errmsg());
    exit(1);
}
```

Create the context object using the context attribute object.

```
if (lbm_context_create(&ctx, cattr, NULL, NULL) == LBM_FAILURE) {
    fprintf(stderr, "lbm_context_create: %s\n", lbm_errmsg());
    exit(1);
}
```

Delete the context attribute object, which is no longer needed.

```
lbm_context_attr_delete(cattr);
```

5.3 Create the Sources

A client application that sends messages must have at least one source object.

When you create a source, configure the attributes in a topic attribute object. After you create the source, you can delete the topic attribute object. The following code fragments show source coding concepts.

Create a topic attribute object with the current default configuration.

```
if (lbm_src_topic_attr_create(&tattr) == LBM_FAILURE) {
    fprintf(stderr, "lbm_src_topic_attr_create: %s\n", lbm_errmsg());
    exit(1);
}
```

Set configuration options that must be unique or permanent for the source. The following example sets configuration option transport to BROKER.

```
if (lbm_src_topic_attr_str_setopt(tattr, "transport", "BROKER") == LBM_FAILURE) {
    fprintf(stderr, "lbm_src_topic_attr_str_setopt:transport: %s\n", lbm_errmsg());
    exit(1);
}
```

Create a source topic object using the context and topic attribute object. The topic string becomes the queue name.

```
if (lbm_src_topic_alloc(&topic, ctx, "QueueName", tattr) == LBM_FAILURE) {
    fprintf(stderr, "lbm_src_topic_alloc: %s\n", lbm_errmsg());
    exit(1);
}
```

Delete the topic attribute object, or retain it for subsequent source topic allocation.

```
lbm_src_topic_attr_delete(tattr);
```

Create the source object using context and source topic object. An example of the function "app_src_callback" is shown below.

```
if (lbm_src_create(&src, ctx, topic, app_src_callback, NULL, NULL) == LBM_FAILURE) {
    fprintf(stderr, "lbm_src_create: %s\n", lbm_errmsg());
    exit(1);
}
```

If needed, set the message properties for the message. In this example we set message property Price to a value of 170 and message property Quan to 49.

```
lbm_src_send_ex_info_t info; int32_t value;
int ret;
.../
* setup */
memset(&info, 0, sizeof(lbm_src_send_ex_info_t));
info.flags = LBM_SRC_SEND_EX_FLAG_PROPERTIES;
lbm_msg_properties_create(&info.properties);/

* set user defined message properties */
value = 170;
ret = lbm_msg_properties_set(info.properties, "Price",
                             (void *) &value,
                             LBM_MSG_PROPERTY_INT, sizeof(int32_t));
assert(ret==LBM_OK);

value = 49;
ret = lbm_msg_properties_set(info.properties, "Quan",
                             (void *) &value, LBM_MSG_PROPERTY_INT,
                             sizeof(int32_t));
assert(ret==LBM_OK);
```

Send the message and delete the message properties object.

```
lbm_src_send_ex(src, "HelloWorld", 10, 0, &info);
...
lbm_msg_properties_delete(info.properties);
```

Verify that the source has registered with the broker. Verify that the sent message is stable at the broker.

```
int app_src_callback(lbm_src_t *src, int event, void *ed, void *cd) {
    ...
    switch (event) {
        ...
        case LBM_SRC_EVENT_UMQ_REGISTRATION_COMPLETE_EX:
            ... /* handle the registration complete */
            break;
        case LBM_SRC_EVENT_UMQ_MESSAGE_STABLE_EX:
            .... /* handle message stability */
            break;
        }
        ...
    }
    ...
}
```

5.4 Create the Receivers

A client application that receives messages must have at least one receiver object.

When you create a receiver, configure the attributes in a topic attribute object. The following code fragments show receiver coding concepts.

Create a receiver topic object using the context object with the current default configuration. The topic string is the queue name.

```
if (lbm_rcv_topic_lookup(&topic, ctx, "QueueName", NULL) == LBM_FAILURE) {
    fprintf(stderr, "lbm_rcv_topic_lookup: %s\n", lbm_errmsg());
    exit(1);
}
```

Create the receiver object using the context and topic object.

```
if (lbm_rcv_create(&rcv, ctx, topic, app_rcv_callback, NULL, NULL) == LBM_FAILURE) {
    fprintf(stderr, "lbm_rcv_create: %s\n", lbm_errmsg());
    exit(1);
}
```

Ultra Messaging receives a message and then delivers the message to the application. The message might have message properties. After the callback delivers the message and returns, Ultra Messaging automatically tells the broker that the message is consumed.

```
/
* Message callback function provided at Receiver Create */
int app_rcv_callback(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd) {
    ...
    switch (msg->type) {
    case LBM_MSG_DATA:
        if (msg->properties != NULL) {
            int ret;
            lbm_msg_properties_iter_t *prop_iter=NULL;

            ret = lbm_msg_properties_iter_create(&prop_iter);
            assert(ret==LBM_OK);

            ret = lbm_msg_properties_iter_first(prop_iter, msg->properties);
            while (ret == LBM_OK) {
                if (strcmp("Price", prop_iter->name) == 0) {
                    assert(prop_iter->type==LBM_MSG_PROPERTY_INT);
                    printf("Price is %d\n", (int32_t) *((int32_t *)
                        prop_iter->data));
                }
                else if (strcmp("Quan", prop_iter->name) == 0) {
                    assert(prop_iter->type==LBM_MSG_PROPERTY_INT);
                    printf("Quantity is %d\n", (int32_t) *((int32_t *)
                        prop_iter->data));
                }
                else {
                    printf("Error: Unknown property name: %s, type: %d",
                           prop_iter->name, prop_iter->type);
                }
                ret = lbm_msg_properties_iter_next(prop_iter);
            }
            lbm_msg_properties_iter_delete(prop_iter);
        }
        ... /* implement processing the payload in msg->data */ }
        break;
    ... /* implement processing for other message types */
    default:
        ... /* implement default handler */
        break;
```

```
    }
}
```

5.5 Client Configuration

Ultra Messaging has multiple ways to set configuration options. Determine which methods are most appropriate for your application.

You can set configuration options programmatically or with configuration files. You can set configuration options in the following ways:

- Use a plain text configuration file. You can also assign the file pathname to an environment variable. You can read this file with a function or method.
- Use an XML configuration file for a more flexible approach to setting configuration options. You can point to this file in the same way you use a plain text configuration file. A single XML file can contain options for multiple applications. Moreover, for a single application, you can configure multiple objects such as named contexts and event queues, with different values for the same options. Also, Ultra Messaging Manager (UMM) can manage XML configuration files.
- Set configuration options individually using function or method calls. You can use this technique to override options set from configuration files.

For detailed information about setting configuration options, see **Configuration Overview**.

5.6 Sample Applications

The UMQ package includes sample applications for testing and educational purposes.

- umqrcv (source at `umqrcv.c`)
- umqsrc (source at `umqsrc.c`)

For descriptions and argument information on example applications, as well as source code for all languages, see the `C Examples` source code, `Java Examples` source code, or `.NET Examples` source code documentation.

5.7 Developing Client Applications for JMS

You can write messaging applications with the Java Message Service (JMS) API and programming model.

Use the Oracle JMS API to develop or port Java messaging applications written according to the JMS specification. The Oracle JMS Specification 1.1 contains requirements and guidelines for developing JMS- based Java messaging applications.

You can write Ultra Messaging and JMS applications that interoperate through the broker by setting and getting message properties.

5.7.1 Message Types

The JMS message consists of a header and body. You can set a JMS body type in the application and optionally identify the body type in a header field.

The following table shows JMS message types and their Ultra Messaging message property JMS_UM_Message↩ Type numeric values:

JMS Message Type	`JMS_UM_MessageType` Value
TextMessage	0
BytesMessage	1
MapMessage	2∗
Message	3
ObjectMessage	4∗
StreamMessage	5∗

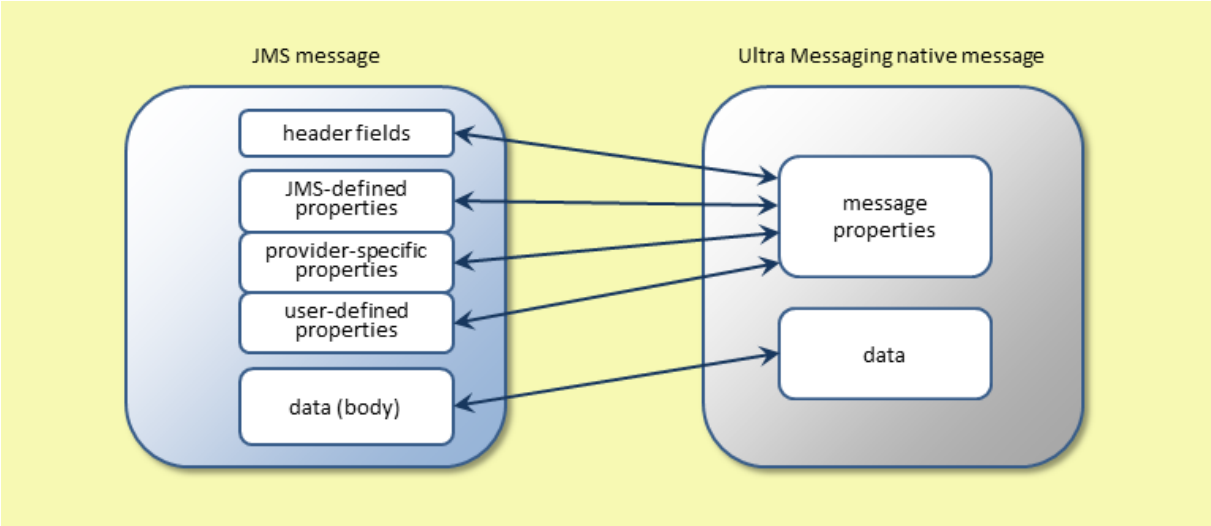
- Not supported in Ultra Messaging queuing. Ultra Messaging handles this JMS message type as a Bytes↩ Message.

5.8 Message Components

When a broker delivers a message to an Ultra Messaging receiver, Ultra Messaging maps JMS message properties and header fields to Ultra Messaging message properties. When an Ultra Messaging source application sends a message to a broker, Ultra Messaging maps message properties to JMS message properties and header fields.

JMS header fields, JMS-defined properties, provider-specific properties, and user properties map to message properties in an Ultra Messaging message. JMS message data in the message body maps to data in an Ultra Messaging message.

The following figure shows how a JMS message maps to an Ultra Messaging message:



A JMS message header consists of the following header fields:

- JMSDestination
- JMSDeliveryMode
- JMSMessageID
- JMSTimestamp
- JMSCorrelationID
- JMSReplyTo
- JMSRedelivered
- JMSType
- JMSExpiration
- JMSPriority

5.9 JMS Message Properties

Ultra Messaging transports JMS message properties of the following categories:

5.9.1 JMS-defined Properties

The JMS Specification defines these properties and reserves the use of their names. JMS-defined properties use a "JMSX" prefix.

5.9.2 Provider-specific Properties

Ultra messaging defines and reserves the following provider-specific properties:

- JMS_UM_MessageType
- JMS_UM_Regid
- JMS_UM_Stamp
- JMS_UM_SQN

5.9.3 User Properties

Properties that you define for applications. You can use User Properties as Message Selectors.

Chapter 6

Fault Tolerance

UMQ uses multiple techniques to ensure that messages reach their destinations. These techniques include the storage and verification of messages and node redundancy.

Ultra Messaging queuing systems use message persistence to confirm to the source application that a message is stored on a broker queue. The source no longer needs to hold a copy of that message. Message durability holds the message in the queue until the receiver application sends a consumption acknowledgement to the queue. These techniques help ensure messages reach their destination if a connection, source, or receiver fails.

An Ultra Messaging high-availability queuing system uses three broker nodes, which can all reside on separate machines. One broker actively handles message traffic while the other two serve as hot standby brokers. If the active broker or broker host fails, messaging can continue relatively uninterrupted.

6.1 Message Reliability

To ensure that messages reach receiver applications, Ultra Messaging uses message stability, persistence, and consumption reporting.

6.1.1 Message Stability

Message stability is an indication that the broker has received a message from the source.

The broker confirms receipt of each message it receives from the sending client application. When at least two nodes acknowledge message stability, the active broker sends a stability acknowledgment back to the source.

Typically, after a source sends a message, it retains that message because it might need to retransmit the message. When the source receives a stability acknowledgment, it can then discard the message. During the period after the source sends the message until the source receives the stability acknowledgment, the message status is in flight. If a broker or broker connection fails, the source application can retain the in-flight messages for retransmission.

6.1.2 Message Consumption

Message consumption is an indication that the receiving client application has processed a message from the broker.

The receiving client application receives and processes each message from the broker. After the callback delivers the message and returns, Ultra Messaging automatically tells the broker that the message is consumed.

After a broker sends a message, it retains that message because it might need to retransmit the message if the receiver application or connection to the receiver application fails. When the broker receives a message consumption acknowledgment, it can then discard the message.

6.1.3 Message Persistence

By default, queuing sources set all messages to be persisted. When the broker receives a message that is set for persistence, the broker holds that message in long-term storage until the receiving client application consumes the message.

To set persistence in a message, you must set the JMSDeliveryMode message property for the message. The following example shows a way that you can set a message property to persistent:

```
/
* setup */
memset(&info, 0, sizeof(lbm_src_send_ex_info_t));
info.flags = LBM_SRC_SEND_EX_FLAG_PROPERTIES;
lbm_msg_properties_create(&info.properties);/

* set a user defined message property */
value = 2;
ret = lbm_msg_properties_set(info.properties, "JMSDeliveryMode",
                             (void *) &value, LBM_MSG_PROPERTY_INT,
                             sizeof(int32_t));
assert(ret==LBM_OK);
```

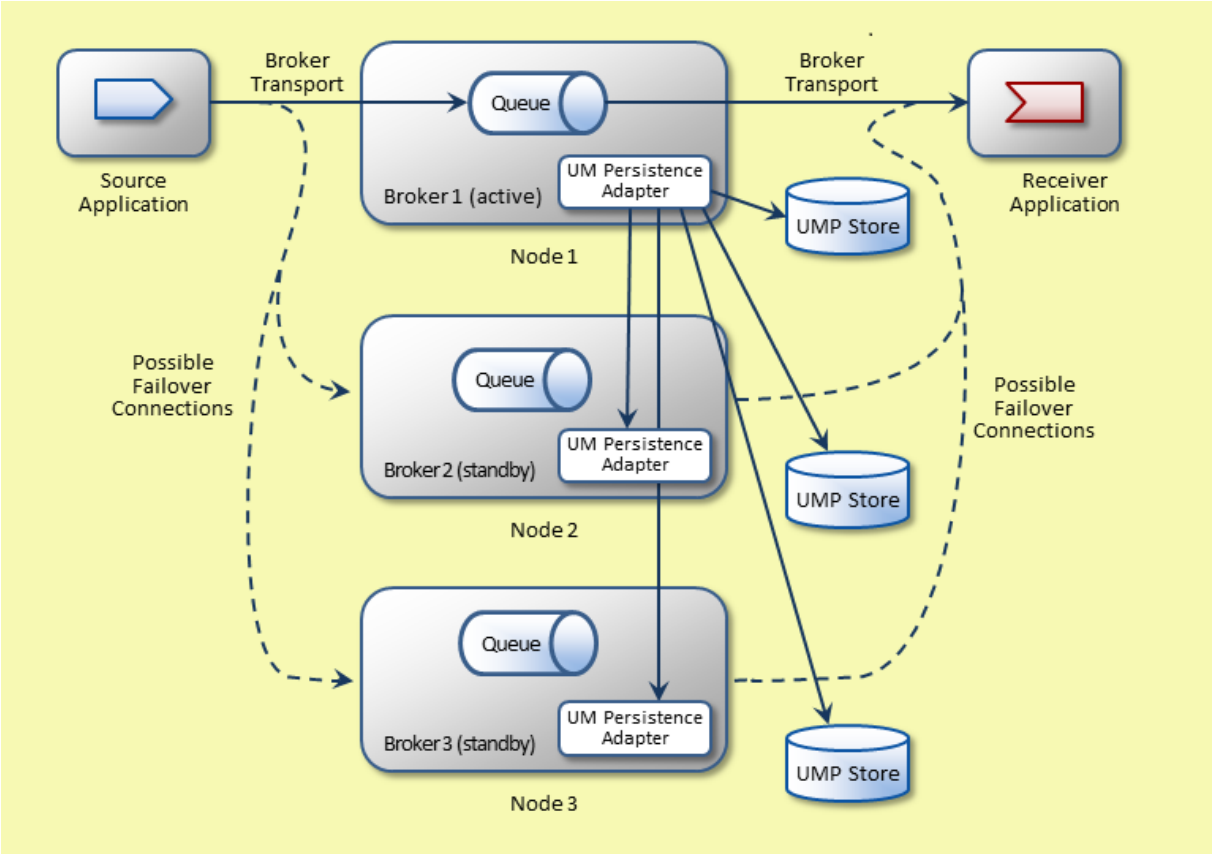
6.2 Node Redundancy

A high availability queuing system uses three host machines on which to run the components of a node. This deployment uses redundancy for processes and for persistence storage.

6.2.1 High Availability

To use configure an Ultra Messaging queuing system for high availability, you must use the queuing three-node deployment. Each node resides on a different host, to eliminate any broker single point of failure.

The following image shows a high-availability queuing system with one source application, one receiver application, and three brokers:



The source application sends a persistent message to the active broker. For each persistent message, the active broker performs the following operations:

1. The active broker receives the persistent message from the source application and attempts to persist the message to disk before adding it to its queue in memory.
2. The active broker requests that its UM persistence adapter persist to disk the status that the message is added to its queue.
3. The UM persistence adapter replicates an add message event to all UM persistence adapters in the standby brokers and itself, and to all UMP stores.
4. The active broker waits until it receives stability from a majority of the UMP stores.
5. The active broker puts the message into its queue and sends a response back to the source application that the message is stable and has been persisted to disk.
6. The broker assigns the message to an eligible receiver and requests that its UM persistence adapter persist to disk the status that the message is assigned.
7. The UM persistence adapter replicates an update message event to the standby UM persistence adapters and to all UMP stores.
8. The active broker waits until it receives stability from a majority of the UMP stores.
9. The active broker sends the message to the receiver application.
10. The receiver application processes the message and then tells the broker that the message is consumed.
11. The active broker requests that its UM persistence adapter persist to disk the status that the message is removed from its queue.
12. The UM persistence adapter replicates a remove message event to the standby UM persistence adapters and to all UMP stores.

13. The active broker waits until it receives stability from a majority of the UMP stores.

14. The active broker removes the message from its queue in memory.

This configuration keeps all three UMP stores up to date with the latest messages and their statuses. If the active node fails, either standby node is ready to continue with the message broker service, and the standby brokers elect a new active broker.

6.2.2 High Availability System Initiation

A high availability queuing system follows an initiation procedure when you first start the system.

When a high availability queuing system first starts, it must determine which node is the active node and which nodes are the standby nodes. The brokers do this through an active-broker election process. The queuing system performs the following initiation actions:

1. All nodes start.

2. The nodes elect an active broker, which opens a transport connection for client applications. The other brokers become standby brokers and do not open connections.

3. Client applications start.

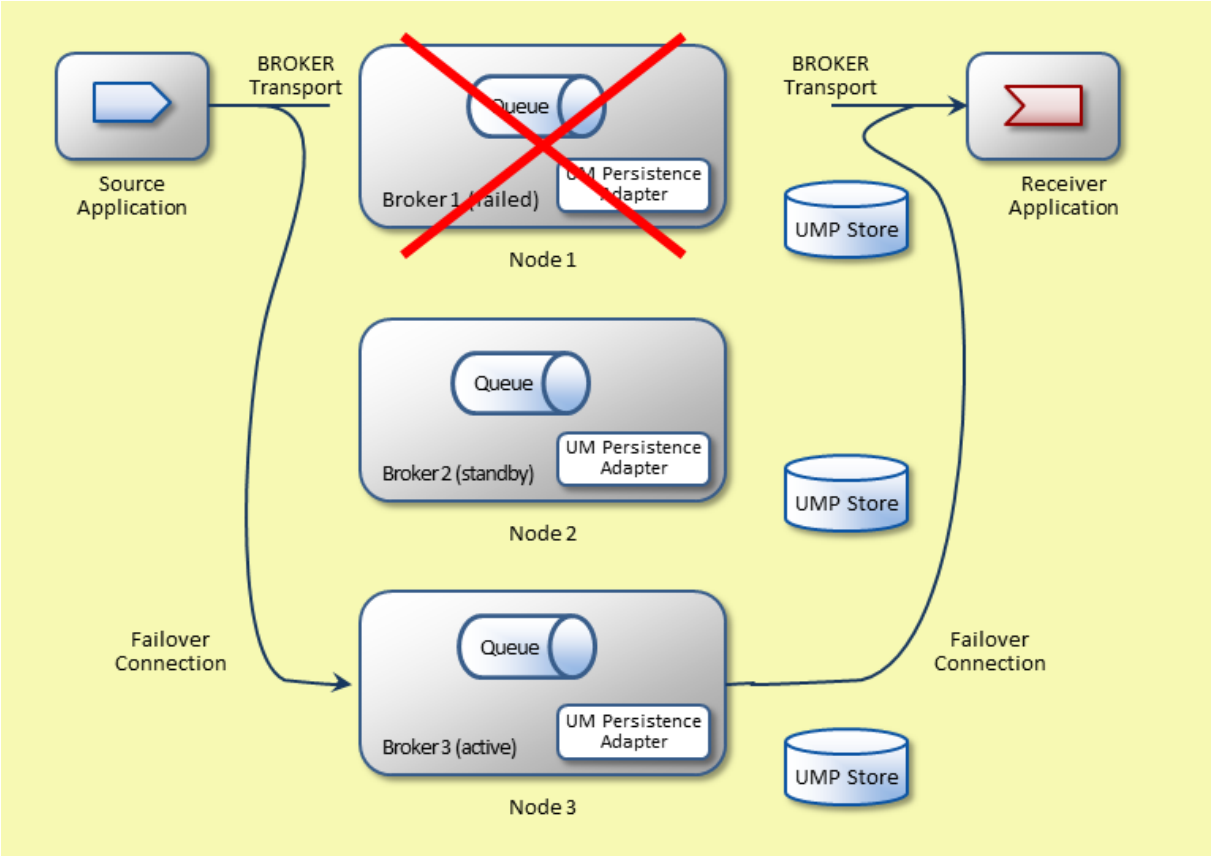
4. Client applications try broker addresses from the list of addresses to determine which broker is active.

5. Client applications initiate and complete registration with the active broker.

6.3 Node Failover

If an active broker fails, Ultra Messaging starts a reelection process to elect a new active broker.

The following image shows the change after an active broker fails and the queuing system elects a new active broker:



In this example, Broker1 has failed and disconnected from the client applications. The client applications then try all IP and port addresses on their broker option list until they find an open socket to connect to. If a client application does not find an active broker, it waits and then tries the list again. This cycle continues until the system elects a new active broker.

After Broker1 failure, Ultra Messaging Broker3 to be the new active broker. The clients then connect to, and register with, the newly active broker. After registration, the clients can resume messaging.

Active and standby brokers send a periodic heartbeat message to each other. If standby brokers do not receive any heartbeat message from the active broker then the standby brokers elect a new active broker.

If an active broker does not receive heartbeat messages from any standby brokers, the active broker becomes a standby broker. The queuing system fails when a quorum of brokers no longer exists.

Chapter 7

Ultra Load Balancing (ULB)

Use Ultra Load Balancing (ULB) to evenly distribute a message load to receiving applications without a broker between sending and receiving applications.

A ULB source sends messages on a UM topic. The source also sends receiver assignment control information. This control information identifies which receiver is to process each message. Receivers unicast message consumption reports back to the source.

ULB sources and receivers use standard Ultra Messaging contexts and do not connect to brokers.

For information on ULB configuration options, see **Ultra Messaging Queuing Options**, paying special attention to options with "ulb" in the name.

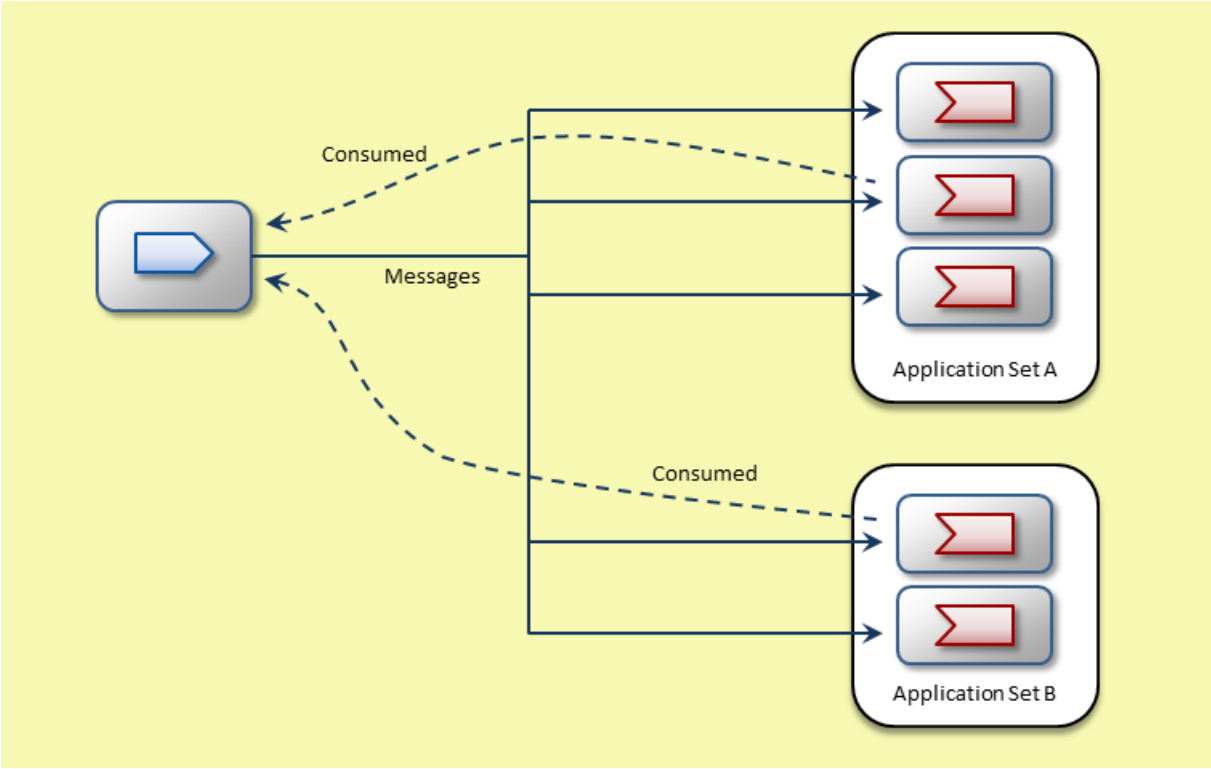
Attention

> The ULB feature is not compatible with setting **ordered_delivery (receiver)** to **"0"** (arrival order without reassembly).

7.1 Application Sets and Receiver Type IDs

You must group ULB receivers into one or more groups called application sets. Within an application set, a ULB source assigns each sent message to only one receiver. If multiple applications sets are defined, then each message is assigned to one receiver *in each set*. I.e. multiple copies of each message are sent, one for each application set.

The following diagram shows how an ULB source disseminates messages to receivers in application sets:



Subscribing applications create ULB receivers that register with a ULB source. The source adds each receiver to an application set according to a *receiver type ID* which is supplied by the subscribing application. The source's configuration defines which receivers are added to which application sets using the configuration option **umq_ulb_application_set (source)**.

Application Sets are identified with an integer between 0 and 65535. The sets must be numbered consecutively, starting from 0. In the diagram above, application set "A" would be 0, and "B" would be 1.

Receiver Type IDs are arbitrary 32-bit integers selected and managed by the application designer. There is no requirement for these IDs to be consecutively numbered. When a receiver registers with the source, the ID is used by the source to determine various configuration options to be associated with that receiver, such as which application set the receiver belongs in. Here are the configuration options that the source associates with receivers by their type IDs:

- **umq_ulb_application_set (source)**
- **umq_ulb_receiver_portion (source)**
- **umq_ulb_receiver_priority (source)**
- **umq_ulb_receiver_events (source)**

For example, in the diagram above the source might be an order entry system, load balancing orders across a set of three execution engines (application set A), and also load balancing those same orders across a set of two logger processes (application set B). We could define execution engines to be type 10 and the loggers to be type 20. The source would be configured as:

```
source umq_ulb_application_set 0:10;1:20
```

Three execution engines would start up, each setting **umq_receiver_type_id (receiver)** to 10. Two logger processes would start up, each setting **umq_receiver_type_id (receiver)** to 20. Thus, each order will be sent to two receivers: one of the execution engines and one of the loggers.

A ULB source sends both the message data and the assignment control information on the same transport session to all receivers. The source also sends assignment data so that the assigned receiver in each application set processes the message. The receivers not assigned ignore the message. When the assigned receivers process their messages, they unicast Consumed reports back to the source.

7.2 Load Balancing

ULB differs from brokered queues regarding assignment uniformity (fairness). Multiple ULB sources to the same topic do not coordinate their assignments with each other. It is possible for multiple ULB sources to all send their next message to the same receiver.

Ultra Load Balancing governs fairness between sources by using the following information:

- receiver feedback information
- receiver portion size
- configuration option **umq_ulb_application_set_load_factor_behavior (source)**

7.2.1 ULB Performance

ULB is built on top of standard UM sources and receivers. As such, all receivers receive all messages sent to the topic, even ULB receivers. The normal load-balancing queuing semantics are implemented above the transport, so that only the assigned receiver delivers a particular message to the application; the other un-assigned receivers discard the message.

For multicast-based data transport (LBT-RM), there is no performance penalty to sending each message to all receivers. However, for unicast-based data transports (e.g. LBT-RU, LBT-TCP), the normal source semantic of delivering each message to all receivers represents an inefficiency and limits the maximum sustainable throughput. As receivers are added to a ULB topic, the efficiency continues to drop due to unnecessary sends of messages to unassigned receivers, thus lowering the maximum sustainable throughput further with each added receiver.

ULB Source-Side Filtering

Starting with UM version 6.12, the Source-Side Filtering feature has been enhanced to support load-balancing queue semantics. Configure the ULB source with the option **transport_source_side_filtering_behavior (source)** set to `"ulb"`. This causes the unicast source to send a particular message to *only* the assigned receiver. This reduces work for both the unicast source and for the un-assigned receivers. As receivers are added, the maximum sustainable throughput does not significantly drop.

To benefit from this feature, receivers *must* be configured for arrival order delivery by setting **ordered_delivery (receiver)** to `-1`. Receivers without that setting will work correctly with the source, but will not benefit from the ULB Source-Side Filtering feature. I.e. receivers configured for ordered delivery value `1` (the default) will continue to receive all messages sent, and discard the non-assigned messages.

A ULB source must be at UM version 6.12 or beyond to enable ULB Source Side Filtering. That source is fully interoperable with receivers at UM versions before 6.12, but the older receivers will not benefit from the ULB Source-Side Filtering feature. I.e. pre-6.12 receivers will continue to receive all messages sent, and discard the non-assigned messages.

7.3 Ultra Load Balancing Flight Size

Ultra Load Balancing uses a flight size mechanism to limit the number of messages sent but not consumed, and blocks a source or warns when a send exceeds the configured flight size. Set Ultra Load Balancing with configuration option **umq_ulb_flight_size (source)**.

You can set configuration option **umq_ulb_flight_size_behavior (source)** to either of the following behaviors:

- Block any sends that would exceed the flight size.
- Continue to allow sends while notifying the sending application.

Ultra Load Balancing considers a message in-flight until the source receives the **MSG_COMPLETE** event, based on a message consumption or a message timeout from all assigned receivers.

7.4 Indexed Ultra Load Balancing

Indexed Ultra Load Balancing is a way to group messages and ensure that one receiver processes all messages in the group in the order they are sent. The source uses an index to assign the messages in a group. An index is an application-defined 64-bit unsigned number or free-form string.

If you must process a set of messages in order, it can be more efficient for one receiver to process the messages. You can set the same index in all related messages to ensure that they all go to one receiver.

Send messages with an index using send call **lbm_src_sendv_ex()** that includes a pointer to **lbm_umq_index_↩info_t** in **lbm_src_send_ex_info_t**.

By default, all receivers are eligible for index assignment. When the source assigns the first message with a particular index to a receiver in each application set, the source assigns subsequent messages with that same index to those same receivers.

Note that because Ultra Load Balancing sources make assignment decisions independently, multiple sources that send on the same index might assign the index to different receivers.

7.5 Total Message Lifetimes for Ultra Load Balancing

You can set a total message lifetime for Ultra Load Balancing applications in different ways.

An Ultra Load Balancing message total lifetime starts when the source sends the message, and ends when the source receives a consumption report or times out. In each case, you set the timeout to a value in milliseconds. If the message times out, the source reclaims the memory.

By default, there is no message lifetime limit. Messages must be consumed.

A timeout can be set with the configuration option **umq_ulb_application_set_message_lifetime (source)**.

Also, it is possible to set a timeout on a per-message basis using the **lbm_src_send_ex()** API. Set **lbm_src_↩send_ex_info_t_stct::umq_total_lifetime** to point at a **lbm_umq_msg_total_lifetime_info_t** structure, and fill in the **lbm_umq_msg_total_lifetime_info_t_stct::umq_msg_total_lifetime** field.

If you set a timeout using both the configuration option and the API, the API overrides the configured setting.

Chapter 8

UMQ Events

Ultra Messaging queuing applications generate context, source, and receiver events based on specific actions or results.

8.1 Context Events

Ultra Messaging generates context events that your client application can listen for and respond to.

You can use context events to determine the state of the context connection.

Ultra Messaging generates the following brokered context events:

Registration Success (LBM_CONTEXT_EVENT_UMQ_REGISTRATION_SUCCESS_EX)

and **Registration complete** (LBM_CONTEXT_EVENT_UMQ_REGISTRATION_COMPLETE_EX)
When a context connects to a broker, Ultra Messaging generates these events.

Registration Error (LBM_CONTEXT_EVENT_UMQ_REGISTRATION_ERROR)

If a context loses connection with a broker, Ultra Messaging generates this event.

8.2 Source Events

Ultra Messaging generates source events that the client application can listen for and respond to.

You can use source events to determine the state of the source and messages. For example, when you write a callback function to respond to source events, you can check for a registration complete event and start sending messages only after you receive the event.

When you use a brokered context, Ultra Messaging generates the following source events:

Registration complete (LBM_SRC_EVENT_UMQ_REGISTRATION_COMPLETE_EX)

A source must be registered on the broker before the broker can process messages from the source. When you create a source, you trigger a source registration. A registration complete source event indicates that the source is ready to send messages to the broker. Ultra Messaging generates an error if you send a message before the source is registered with a broker.

Message stable (LBM_SRC_EVENT_UMQ_MESSAGE_STABLE_EX)

Ultra Messaging generates a message stable event when a message reaches the broker and is stable. A message is stable when the broker has successfully processed the message based on the message queue configuration. If you receive a message stable event for a message, you do not have to resend the message even if the source is disconnected from the broker.

If the source is disconnected from the broker before you receive a message stable event for a message, you might need to resend the message. Ultra Messaging does not automatically resend a message upon disconnection. The client application must include a mechanism to process messages that do not reach the broker.

Message Reclaimed (LBM_SRC_EVENT_UME_MESSAGE_RECLAIMED_EX)

Ultra Messaging generates a message reclaimed event for each message sent. If the forced message reclaim flag is set (**SRC_EVENT_UME_MESSAGE_RECLAIMED_EX_FLAG_FORCED**), there is no guarantee that the message is stable.

Message Reclaimed (LBM_SRC_EVENT_UME_MESSAGE_RECLAIMED)

This event is the same as the **LBM_SRC_EVENT_UME_MESSAGE_RECLAIMED_EX** event, except that it does not include a forced message reclaim flag.

Registration error (LBM_SRC_EVENT_UMQ_REGISTRATION_ERROR)

If the source is disconnected from the broker, Ultra Messaging generates a registration error event to indicate that the source is no longer registered with the broker. For example, if the broker shuts down and the source loses its connection to the broker, Ultra Messaging generates a registration error event and any message that you send or is in flight does not reach the broker. You must wait for a registration complete event before you send more messages to the broker.

In a high availability multi-node deployment, if the active broker shuts down, Ultra Messaging elects one of the standby brokers as the new active broker. The source connects to the new active broker by default. Ultra Messaging generates a registration error event when the source loses connection to the previous active broker and a registration complete event when the source successfully registers with the new active broker.

Message ID (LBM_SRC_EVENT_UMQ_MESSAGE_ID_INFO)

For each message a source sends, Ultra Messaging generates an event that contains the message ID.

To enable this event, set the **LBM_SRC_SEND_EX_FLAG_UMQ_MESSAGE_ID_INFO** flag of the **lbm_src_send_ex_info_t** object passed on the send call.

Message Sequence Number (LBM_SRC_EVENT_SEQUENCE_NUMBER_INFO)

For each message a source sends, Ultra Messaging generates an event that contains the message sequence number.

To enable this event, set the **LBM_SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO** flag of the **lbm_src_send_ex_info_t** object passed on the send call.

Flight Size Notification (LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION)

Ultra Messaging can generate a flight size event when the number of in-flight messages exceeds or drops under the configured flight size. To enable this event, set configuration option **umq_flight_size_behavior (context)** to **notify**. Also, recast the event data pointer as type **lbm_src_event_flight_size_notification_t**.

If the number of in-flight messages exceeds the configured flight size, the **lbm_src_event_flight_size_notification_t::state** value is **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION_STATE_OVER**. If the number of in-flight messages drops below the configured flight size, the **lbm_src_event_flight_size_notification_t::state** value is **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION_STATE_UNDER**.

Source Wakeup Notification (LBM_SRC_EVENT_WAKEUP)

Ultra Messaging can generate a wakeup event when a source send operation, blocked because the number of in-flight messages exceeded the configured flight size, is able to send again. To enable this event, set configuration option **umq_flight_size_behavior (context)** to **block**. Also, recast the event data pointer as pointer type **lbm_src_event_wakeup_t** $*$.

When a source becomes unblocked, the **lbm_src_event_wakeup_t::flags** value is **LBM_SRC_EVENT_W↩ AKEUP_FLAG_NORMAL**.

8.3 Receiver Events

Ultra Messaging generates receiver events that the client application can listen for and respond to.

You can use receiver events to determine the state of the receiver and messages.

When you use brokered contexts, Ultra Messaging generates Registration complete event (**LBM_MSG_UMQ_R↩ EGISTRATION_COMPLETE_EX**).

A receiver must be registered on the broker before the broker can send messages to the receiver. When you create a receiver, you trigger a receiver registration. Upon registration with a broker, Ultra Messaging sends to the client application an instance of the **lbm_msg_umq_registration_complete_ex_t** object. This object contains the broker name, IP address, and port.

Registration error (LBM_MSG_UMQ_REGISTRATION_ERROR)

Ultra Messaging generates this event if any of the following actions occurs:

1. A broker does not connect with a receiver.
2. A brokered receiver has lost connection to a broker.

The event message contains the broker name, IP address, and port.

Beginning of Index (LBM_MSG_UMQ_INDEX_ASSIGNED_EX)

When a receiver detects a new index from the broker, Ultra Messaging generates this event. The event message contains broker and index information.

Release of Index (LBM_MSG_UMQ_INDEX_RELEASED_EX)

When a receiver receives the last message in a group of index-queued messages, Ultra Messaging generates this event. The event message contains broker and index information.

8.4 Event Changes

UMQ 6.8 and beyond does not generate all the events that pre-6.8 versions generated.

The following list shows events from UMQ versions earlier than 6.8 that UMQ 6.8 queuing does not generate:

- **LBM_MSG_UMQ_INDEX_ASSIGNMENT_ERROR**
- **LBM_MSG_UMQ_INDEX_ASSIGNMENT_ELIGIBILITY_START_COMPLETE_EX**
- **LBM_MSG_UMQ_INDEX_ASSIGNMENT_ELIGIBILITY_STOP_COMPLETE_EX**
- **LBM_MSG_UMQ_INDEX_ASSIGNMENT_ELIGIBILITY_ERROR**
- **LBM_MSG_BOS**
- **LBM_MSG_EOS**
- **LBM_MSG_NO_SOURCE_NOTIFICATION**
- **LBM_MSG_UMQ_DEREGISTRATION_COMPLETE_EX**
- **LBM_CONTEXT_EVENT_UMQ_INSTANCE_LIST_NOTIFICATION**

Chapter 9

Configuration Option Changes

As of Ultra Messaging Queuing 6.8, some configuration options are new, some function differently, and some are deprecated.

9.1 New Configuration Options

As of Ultra Messaging Queuing 6.8, the following new configuration options are available:

- **broker (context)**

9.2 Changed Configuration Options

As of UMQ 6.8, you use some configuration options differently.

You can use the following configuration options for Ultra Load Balancing only:

- **umq_session_id (context)**
- **umq_delayed_consumption_report_interval (receiver)**
- **umq_hold_interval (receiver)**
- **umq_index_assignment_eligibility_default (receiver)**
- **umq_receiver_type_id (receiver)**
- **umq_retransmit_request_interval (receiver)**
- **umq_retransmit_request_outstanding_maximum (receiver)**

Configuration option **transport (source)** has an additional value: **broker**.

String	Integer Value	Description
broker	**LBM_SRC_TOPIC_ATTR_TRANSPORT_BR↩ OKER**	Sources send messages to a broker, which manages the messages for consumption.

Option **umq_queue_activity_timeout (context)** has no effect.

Configuration option **umq_queue_registration_id (context)** functions differently and no longer helps to manage duplicate message delivery.

For configuration option **umq_queue_participation (receiver)**, you cannot set value 2 to enable queue browsing with brokered contexts.

See also **Deprecated Options**.

9.3 Deprecated Configuration Options

As of UMQ 6.8, the following configuration options are deprecated:

- **umq_flight_size (context)**
- **umq_flight_size_behavior (context)**
- **umq_msg_total_lifetime (context)**
- **umq_message_retransmission_interval (context)**
- **umq_message_stability_notification (context)**
- **umq_queue_check_interval (context)**
- **umq_queue_name (source)**
- **umq_queue_participants_only (source)**
- **umq_queue_query_interval (context)**
- **umq_require_queue_authentication (context)**
- **umq_retention_intergroup_stability_behavior (context)**
- **umq_retention_intragroup_stability_behavior (context)**
- **umq_retention_intergroup_stability_behavior (source)**
- **umq_retention_intragroup_stability_behavior (source)**

Chapter 10

Deprecated and Unavailable Features

As of Ultra Messaging Queuing 6.8, some queuing features are deprecated and some UMS features are unavailable to brokered contexts. Also, you cannot use all variations of some features.

10.1 Deprecated Features

The following UMQ features are deprecated and also unavailable:

- Source Dissemination
- Parallel Queue Dissemination
- Serial Queue Dissemination
- UMQ Queue Browsing
- Automatic source resubmission on failover
- Numeric index IDs for Indexed Queuing

10.2 Deprecated Functions and Methods

As of UMQ 6.8, the following C API functions are deprecated:

- **lbm_ctx_umq_get_inflight()**
- **lbm_umq_ctx_msg_stable()**
- **lbm_queue_immediate_message()**

The following Java and .NET API classes and methods are deprecated for Ultra Messaging 6.8 or later:

- UMEBlockSrc class
- LBMContext.setUMQInflight() method
- LBMContext.setUMQMessageStable() method

10.3 Unavailable Features

Ultra Messaging queuing applications can use some Ultra Messaging functionality, but some functionality is unavailable with Ultra Messaging queuing.

You cannot use Ultra Messaging queuing with the following Ultra Messaging core functionalities:

- Acceleration - DBL
- Acceleration - UD
- Explicit Batching
- FD Management
- Hot Failover (HF)
- Hot Failover Across Contexts (HFX)
- Implicit Batching
- Late Join
- Multicast Immediate Messaging (MIM)
- Multi-Transport Threads
- Off-Transport Recovery (OTR)
- Request and Response
- Source Side Filtering
- Spectrum
- Transports other than type broker
- Ultra Load Balancing (ULB)
- Ultra Messaging Desktop Services (UMDS)
- Ultra Messaging Spectrum
- Ultra Messaging System Monitoring
- UMCache
- UMP stores
- UM Router or Gateway
- UM SNMP Agent
- Ultra Messaging Wildcard Receivers

You cannot use brokered contexts with the following functions or methods:

- **lbm_rcv_umq_deregister()**

10.4 Limited Ultra Messaging Functionality

Use the following rules and guidelines when you use Ultra Messaging queuing with Ultra Messaging core functionalities:

- When you use queuing, you cannot set configuration option **fd_management_type (context)**.
- Ultra Messaging queuing systems do not fragment messages.