# The Ultra Messaging® Guide for Persistence and Queuing

**Copyright © 2007 - 2014 Informatica**
**March 2014**

**Informatica Ultra Messaging**
**Version 5.3**
**March 2014**
**Copyright (c) 1998-2014 Informatica Corporation. All rights reserved.**
**This software and documentation contain proprietary information of Informatica Corporation and are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright law. Reverse engineering of the software is prohibited. No part of this document may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording or otherwise) without prior consent of Informatica Corporation. This Software may be protected by U.S. and/or international Patents and other Patents Pending.**
**Use, duplication, or disclosure of the Software by the U.S. Government is subject to the restrictions set forth in the applicable software license agreement and as provided in DFARS 227.7202-1(a) and 227.7702-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (OCT 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as applicable.**
**The information in this product or documentation is subject to change without notice. If you find any problems in this product or documentation, please report them to us in writing.**
**Informatica, Informatica Platform, Informatica Data Services, PowerCenter, PowerCenterRT, PowerCenter Connect, PowerCenter Data Analyzer, PowerExchange, PowerMart, Metadata Manager, Informatica Data Quality, Informatica Data Explorer, Informatica B2B Data Transformation, Informatica B2B Data Exchange Informatica On Demand, Informatica Identity Resolution, Informatica Application Information Lifecycle Management, Informatica Complex Event Processing, Ultra Messaging and Informatica Master Data Management are trademarks or registered trademarks of Informatica Corporation in the United States and in jurisdictions throughout the world. All other company and product names may be trade names or trademarks of their respective owners.**
**Portions of this software and/or documentation are subject to copyright held by third parties, including without limitation: Copyright DataDirect Technologies. All rights reserved. Copyright (c) Sun Microsystems. All rights reserved. Copyright (c) RSA Security Inc. All Rights Reserved. Copyright (c) Ordinal Technology Corp. All rights reserved.Copyright (c) Aandacht c.v. All rights reserved. Copyright Genivia, Inc. All rights reserved. Copyright Isomorphic Software. All rights reserved. Copyright (c) Meta Integration Technology, Inc. All rights reserved. Copyright (c) Intalio. All rights reserved. Copyright (c) Oracle. All rights reserved. Copyright (c) Adobe Systems Incorporated. All rights reserved. Copyright (c) DataArt, Inc. All rights reserved. Copyright (c) ComponentSource. All rights reserved. Copyright (c) Microsoft Corporation. All rights reserved. Copyright (c) Rogue Wave Software, Inc. All rights reserved. Copyright (c) Teradata Corporation. All rights reserved. Copyright (c) Yahoo! Inc. All rights reserved. Copyright (c) Glyph & Cog, LLC. All rights reserved. Copyright (c) Thinkmap, Inc. All rights reserved. Copyright (c) Clearpace Software Limited. All rights reserved. Copyright (c) Information Builders, Inc. All rights reserved. Copyright (c) OSS Nokalva, Inc. All rights reserved. Copyright Edifecs, Inc. All rights reserved. Copyright Cleo Communications, Inc. All rights reserved. Copyright (c) International Organization for Standardization 1986. All rights reserved. Copyright (c) ej-technologies GmbH. All rights reserved. Copyright (c) Jaspersoft Corporation. All rights reserved. Copyright (c) is International Business Machines Corporation. All rights reserved. Copyright (c) yWorks GmbH. All rights reserved. Copyright (c) Lucent Technologies. All rights reserved. Copyright (c) University of Toronto. All rights reserved. Copyright (c) Daniel Veillard. All rights reserved. Copyright (c) Unicode, Inc. Copyright IBM Corp. All rights reserved. Copyright (c) MicroQuill Software Publishing, Inc. All rights reserved. Copyright (c) PassMark Software Pty Ltd. All rights reserved. Copyright (c) LogiXML, Inc. All rights reserved. Copyright (c) 2003-2010 Lorenzi Davide, All rights reserved. Copyright (c) Red Hat, Inc. All rights reserved. Copyright (c) The Board of Trustees of the Leland Stanford Junior University. All rights reserved. Copyright (c) EMC Corporation. All rights reserved. Copyright (c) Flexera Software. All rights reserved. Copyright (c) Jinfonet Software. All rights reserved. Copyright (c) Apple Inc. All rights reserved. Copyright (c) Telerik Inc. All rights reserved.**

**This product includes software developed by the Apache Software Foundation (http://www.apache.org/), and/or other software which is licensed under various versions of the Apache License (the "License"). You may obtain a copy of these Licenses at http://www.apache.org/licenses/. Unless required by applicable law or agreed to in writing, software distributed under these Licenses is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licenses for the specific language governing permissions and limitations under the Licenses.**
**This product includes software which was developed by Mozilla (http://www.mozilla.org/), software copyright The JBoss Group, LLC, all rights reserved; software copyright (c) 1999-2006 by Bruno Lowagie and Paulo Soares and other software which is licensed under various versions of the GNU Lesser General Public License Agreement, which may be found at http:// www.gnu.org/licenses/lgpl.html. The materials are provided free of charge by Informatica, "as-is", without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.**
**The product includes ACE(TM) and TAO(TM) software copyrighted by Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University, Copyright (c) 1993-2006, all rights reserved.**
**This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (copyright The OpenSSL Project. All Rights Reserved) and redistribution of this software is subject to terms available at http://www.openssl.org and http://www.openssl.org/source/license.html.**
**This product includes Curl software which is Copyright 1996-2007, Daniel Stenberg, <daniel@haxx.se>. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http://curl.haxx.se/docs/copyright.html. Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.**
**The product includes software copyright 2001-2005 (c) MetaStuff, Ltd. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http://www.dom4j.org/ license.html.**
**The product includes software copyright (c) 2004-2007, The Dojo Foundation. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http://dojotoolkit.org/license.**
**This product includes ICU software which is copyright International Business Machines Corporation and others. All rights reserved. Permissions and limitations regarding this software are subject to terms available at http://source.icu-project.org/repos/icu/icu/trunk/license.html.**
**This product includes software copyright (c) 1996-2006 Per Bothner. All rights reserved. Your right to use such materials is set forth in the license which may be found at http:// www.gnu.org/software/ kawa/Software-License.html.**
**This product includes OSSP UUID software which is Copyright (c) 2002 Ralf S. Engelschall, Copyright (c) 2002 The OSSP Project Copyright (c) 2002 Cable & Wireless Deutschland. Permissions and limitations regarding this software are subject to terms available at http://www.opensource.org/licenses/mit-license.php.**
**This product includes software developed by Boost (http://www.boost.org/) or under the Boost software license. Permissions and limitations regarding this software are subject to terms available at http:/ /www.boost.org/LICENSE_1_0.txt.**
**This product includes software copyright (c) 1997-2007 University of Cambridge. Permissions and limitations regarding this software are subject to terms available at http:// www.pcre.org/license.txt.**
**This product includes software copyright (c) 2007 The Eclipse Foundation. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http:// www.eclipse.org/org/documents/epl-v10.php.**
**This product includes software licensed under the terms at http://www.tcl.tk/software/tcltk/license.html, http://www.bosrup.com/web/overlib/?License, http://www.stlport.org/doc/ license.html, http:// asm.ow2.org/license.html, http://www.cryptix.org/LICENSE.TXT, http://hsqldb.org/web/hsqlLicense.html, http://httpunit.sourceforge.net/doc/ license.html, http://jung.sourceforge.net/license.txt , http://www.gzip.org/zlib/zlib_license.html, http://www.openldap.org/software/release/license.html, http://www.libssh2.org, http://slf4j.org/license.html, http://www.sente.ch/software/OpenSourceLicense.html, http://fusesource.com/downloads/license-agreements/fuse-message-broker-v-5-3- license-agreement; http://antlr.org/license.html; http://aopalliance.sourceforge.net/; http://www.bouncycastle.org/licence.html; http://www.jgraph.com/jgraphdownload.html;**

**http://www.jcraft.com/jsch/LICENSE.txt; http://jotm.objectweb.org/bsd_license.html; . http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231; http://www.slf4j.org/license.html; http://nanoxml.sourceforge.net/orig/copyright.html; http://www.json.org/license.html; http://forge.ow2.org/projects/javaservice/, http://www.postgresql.org/about/licence.html, http://www.sqlite.org/copyright.html, http://www.tcl.tk/software/tcltk/license.html, http://www.jaxen.org/faq.html, http://www.jdom.org/docs/faq.html, http://www.slf4j.org/license.html; http://www.iodbc.org/dataspace/iodbc/wiki/iODBC/License; http://www.keplerproject.org/md5/license.html; http://www.toedter.com/en/jcalendar/license.html; http://www.edankert.com/bounce/index.html; http://www.net-snmp.org/about/license.html; http://www.openmdx.org/#FAQ; http://www.php.net/license/3_01.txt; http://srp.stanford.edu/license.txt; http://www.schneier.com/blowfish.html; http://www.jmock.org/license.html; http://xsom.java.net; and http://benalman.com/about/license/; https://github.com/CreateJS/EaselJS/blob/master/src/easeljs/display/Bitmap.js; http://www.h2database.com/html/license.html#summary; and http://jsoncpp.sourceforge.net/LICENSE.**

**This product includes software licensed under the Academic Free License http://www.opensource.org/licenses/afl-3.0.php), the Common Development and Distribution License (http://www.opensource.org/licenses/cddl1.php) the Common Public License (http://www.opensource.org/licenses/cpl1.0.php), the Sun Binary Code License Agreement Supplemental License Terms, the BSD License (http:// www.opensource.org/licenses/bsd-license.php) the MIT License (http://www.opensource.org/licenses/mit-license.php) and the Artistic License (http://www.opensource.org/licenses/artistic-license-1.0).**

**This product includes software copyright (c) 2003-2006 Joe Walnes, 2006-2007 XStream Committers. All rights reserved. Permissions and limitations regarding this software are subject to terms available at http://xstream.codehaus.org/license.html. This product includes software developed by the Indiana University Extreme! Lab. For further information please visit http://www.extreme.indiana.edu/.**

**This Software is protected by U.S. Patent Numbers 5,794,246; 6,014,670; 6,016,501; 6,029,178; 6,032,158; 6,035,307; 6,044,374; 6,092,086; 6,208,990; 6,339,775; 6,640,226; 6,789,096; 6,820,077; 6,823,373; 6,850,947; 6,895,471; 7,117,215; 7,162,643; 7,243,110, 7,254,590; 7,281,001; 7,421,458; 7,496,588; 7,523,121; 7,584,422; 7676516; 7,720,842; 7,721,270; and 7,774,791, international Patents and other Patents Pending.**

**DISCLAIMER: Informatica Corporation provides this documentation "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of noninfringement, merchantability, or use for a particular purpose. Informatica Corporation does not warrant that this software or documentation is error free. The information provided in this software or documentation may include technical inaccuracies or typographical errors. The information in this software and documentation is subject to change at any time without notice.**

**NOTICES**

**This Informatica product (the "Software") includes certain drivers (the "DataDirect Drivers") from DataDirect Technologies, an operating company of Progress Software Corporation ("DataDirect") which are subject to the following terms and conditions:**

**1. THE DATADIRECT DRIVERS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**

**2. IN NO EVENT WILL DATADIRECT OR ITS THIRD PARTY SUPPLIERS BE LIABLE TO THE END-USER CUSTOMER FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL OR OTHER DAMAGES ARISING OUT OF THE USE OF THE ODBC DRIVERS, WHETHER OR NOT INFORMED OF THE POSSIBILITIES OF DAMAGES IN ADVANCE. THESE LIMITATIONS APPLY TO ALL CAUSES OF ACTION, INCLUDING, WITHOUT LIMITATION, BREACH OF CONTRACT, BREACH OF WARRANTY, NEGLIGENCE, STRICT LIABILITY, MISREPRESENTATION AND OTHER TORTS.**

# Table of Contents

**1. Introduction** ..... **4**
**2. Concepts** ..... **5**
**3. Architectures** ..... **12**
**4. Operational View** ..... **16**
**5. Enabling Persistence** ..... **47**
**6. Demonstrating Persistence** ..... **51**
**7. Designing Persistence Applications** ..... **56**
**8. Enabling Queuing** ..... **88**
**9. Designing Queuing Applications** ..... **90**
**10. Fault Tolerance** ..... **93**
**11. Man Pages** ..... **106**
**12. Configuration Reference for Umestored** ..... **109**
**13. Ultra Messaging Web Monitor** ..... **143**

# 1. Introduction

In addition to high performance streaming, **Ultra Messaging®** also provides persistence and low latency queuing by implementing a configurable daemon that runs persistent stores , queues or both .

## 1.1. Persistence

A system implementing **UMP** persistence comprises any number of sources, receivers, and persistent stores. **Ultra Messaging**'s unique design provides **Parallel Persistence®**, which refers to the ability of a persistent store or stores to run independently of sources and receivers and in parallel with messaging. The persistence store does not interfere with message delivery to receiving applications. **Parallel Persistence** adds several key features missing in other messaging solutions.

- A fault recovery ability
- The capacity to continue operation during specific types of failures

Fault recovery refers to the system's ability to recover from a failure of any system component (source, receiver or store). Under certain circumstances, **Ultra Messaging** can even recover from multiple failures and multiple cascading failures.

## 1.2. Queuing

In addition to the capabilities of **UMP** and UMS, **UMQ** supports brokered and brokerless queuing semantics.

Key features of **UMQ** include the following:

- Once and only once (OAOO) delivery for applications such as clearing and settlement that require each trade be processed only once.
- Application Sets for publishing into multiple logical queues with a single send, making it easy to onboard new applications that require copies of the same data.
- Ultra Load Balancing (ULB) for brokerless non-persistent queuing semantics, with special considerations for message assignment, receiver pacing, and multi-source fairness.
- Java Messaging Service (JMS) support. See the Ultra Messaging JMS Guide for more details.

Queuing semantics are supported only within the same Topic Resolution Domain (TRD), without Gateways or Dynamic Routers in the data path.

## 1.3. Semantic Responsibilities

Brokered and brokerless queuing semantics are orthogonal and independent from the reliable non-persistent and durable persistent streaming semantics provided by UMS and UMP. Differences outlined in the following list:

- Reliable non-persistent streaming (UMS) provides in-order message delivery and gap/loss notification, while applications are running, without load-balanced message delivery.
- Durable persistent streaming (UMP) provides in-order message delivery and gap/loss notification, while applications are running and across restarts, without load-balanced message delivery.
- Brokered queuing (UMQ) provides load-balanced message delivery, while applications are running and across restarts, without message ordering and without gap/loss notification.
- Brokerless queuing (UMQ) provides load-balanced message delivery, while applications are running, without message ordering and without gap/loss notification.

Regardless of the messaging semantic chosen, applications assume the following responsibilities to the extent applicable for their use case:

- Application resubmission (re-sending) of in-flight messages after source application restart
- Handling potentially duplicate messages from messaging layer or application resubmission
- Marking application resubmitted messages as such if needed to support duplicate handling
- Detecting stale data, for example, by using synchronized timestamps
- Source and receiver application failover and state reconstruction

Queuing is recommended for cases where load-balanced message delivery is required and out-of-order message delivery and message loss without notification are acceptable, i.e., message delivery order and message loss are either unimportant or are handled by applications (inclusive of application-level sequencing and gap detection, or equivalent). For example, queuing is well-suited for load-balanced request/response with bidirectional topics and application-level retries.

When designing queuing applications, it is important to consider whether message processing is stateless (any receiver can process any message) or requires local state (only certain receivers can process certain messages). If local state is required, the recovery strategy for receiver hardware failure must consider how to rebuild that local state. Receivers with local state may be better served by durable persistent streaming rather than queuing.

# 2. Concepts

This section illuminates important **UMP** and **UMQ** concepts and features.

- *Persistence*
- *Queuing Features*

Contained in **UMP** and **UMQ** are all of the features and capabilities of **Ultra Messaging**'s high performance, message streaming. This document explains persistence and queuing capabilities only. For specific information about **Ultra Messaging**'s high performance streaming, see **Ultra Messaging Concepts** (../Design/index.html).

Also available to **UMP** and **UMQ** is the **Ultra Messaging Manager**. **UMM** provides a GUI that simplifies the creation of **UM** XML configuration files and also allows you to assign application configurations to specific users, also created in the UMM GUI. The UMM Daemon runs this feature, offering a **UMM** GUI API to support custom GUIs and uses a MySQL database to store configurations. See the Ultra Messaging Manager Guide (../UMM/index.html).

## 2.1. Persistence

In discussing **UMP** , we refer to specific recovery from the failures of sources, receivers, and persistent stores. Failed sources can restart and resume sending data from the point at which they stopped. Receivers can recover from failure and begin receiving data from the point immediately prior to failure. This process is sometimes called durable subscription. Persistent stores can also be restarted and continue providing persistence to the sources and receivers that they serve. **UMP** is not designed to address ongoing, corrupting agents. Rather, if one of its components fails, the design of **UMP** 's persistence enables it to continue supporting its ongoing operations at some level.

The default mode of **UMP** Persistence is Source-paced Persistence (SPP). In this mode, the consumption of messages by receivers does not impact the rate a **UM** source can send messages. Sources send messages simultaneously to receivers and the persistent store. (See *Normal Operation*.) Receiver-paced Persistence (RPP) is the second mode. In RPP, sources also send messages to receivers and the persistent store in parallel, but the store retains RPP messages until all RPP receivers acknowledge consumption. In addition, sources can be slowed to ensure that the store is not overrun with messages resulting in messages being dropped and not delivered to all RPP receivers. (See *RPP Normal Operations*.)

### 2.1.1. Persistent Store

**UMP** uses a daemon to persist source and receiver state outside the actual sources and receivers themselves. This is the **UMP** Persistent Store. The store can persist state in memory as well as on disk. State is persisted on a per-topic, per-source basis by the store. **UMP** stores need not be a single entity. For fault tolerance purposes, it is possible to configure multiple stores in various ways. See *Adding the UMP Store to a Source*, *UMP Stores*, *Store Configuration Considerations*, *Man Pages* and *Configuration Reference for Umestored*.

### 2.1.2. Registration Identifier

**UMP** identifies sources and receivers with Registration Identifiers, also called Registration IDs or RegIDs. A RegID is a 32-bit number that uniquely identifies a source or a receiver to a store. This means that RegIDs are also specific to a store and can be reused between individual stores, if needed. No two active sources or receivers can share a RegID or use the same RegID at the same time. This point is critical: since **UMP** enables your application to use and

handle RegIDs very freely, you must use RegIDs carefully to avoid destructive results. See *Adding Fault Recovery with Registration IDs* and *Registration Identifiers*. RegIDs can also be managed easily with the use of Session IDs. See *Managing RegIDs with Session IDs*.

### 2.1.3. Delivery Confirmation

**UMP** provides feedback to sources upon notification that a receiver has consumed a given piece of data, in other words, that it has received and processed a message. This feedback is called Delivery Confirmation. See also *Confirmed Delivery* and *Source Message Retention and Release*.

### 2.1.4. Release Policy

Sources and persistent stores retain data according to a release policy, which is a set of rules that specifies when a message can be reclaimed. Each rule would allow any message that complies with the rule to be reclaimed. However, a message must comply with all rules before it can be reclaimed. Conversely, any message not complying with all rules will not be reclaimed. A source or store retains messages until its retention policy dictates the message may be removed. Sources and stores use slightly different retention policies based on their individual roles. For more information on retention policies, see *Source Message Retention and Release*.

### 2.1.5. Message Stability

Sources send messages to both receivers and to stores. Messages become stable once the message has been persisted at the store or a set of stores. The number of messages that can be sent by a source has no relation to the number of its messages that have been stabilized unless *UMP Flight Size* is enabled. In addition, **UMP** informs the application when messages are stabilized, enabling the application to take any desired action. See *Source Message Retention and Release*.

Publishing messages to a store is a coordinated hand-off between the publishing application and the store. The store assumes responsibility for delivering a message only when the publisher is informed that the message is *stable*. Stability refers to the store having a copy of the message in memory and/or on disk, depending on configuration. Until the publisher is informed that a message is stable, it may be lost upon restart. Messages may be delivered without the publisher being informed that they were stable. Upon restart, the publisher is expected to send again any messages previously sent that were not known to be stable. Since the store cannot differentiate between new messages and messages sent again upon publisher restart, the application is responsible for marking messages as sent again in some manner (e.g., by setting a flag in message content or properties), if required for downstream duplicate checking.

### 2.1.6. Round-Robin Store Failover

Stores can also experience failures from which they may or may not recover. A source can be configured to move to a second store if the first store fails and can not recover in time. Round-robin store behavior describes the behavior of a source moving through a list of stores, using a single store at any one time, with several specified backups available to it in case the single store fails.

See also *Sources Using Round-Robin Store Configuration* and *Round-Robin Store Usage*

### 2.1.7. Quorum/Consensus Store Failover

In addition to a source being configured for round-robin store behavior, several stores can be configured for simultaneous operation. In this situation, a single store or even a handful of stores can fail without impacting the source and receivers. As long as a quorum of the configured stores is accessible, messaging operation generally continues uninterrupted. (**UMP** defines a Quorum as a majority.)

See also *Sources Using Quorum/Consensus Store Configuration*, *Quorum/Consensus Store Usage*, *Quorum/Consensus - Single Location Groups* and *Quorum/Consensus - Mixed Location Groups*.

## 2.2. Queuing Features

A queue may be persistent or may be volatile in nature. Receiver message processing follows a once-and-only-once (OAOO) semantic where each message is only processed by a single receiver of the application set. The following concepts are integral to **UMQ**.

### 2.2.1. Source Streaming

Sources may send and have in flight several messages to the queue at the same time. This provides some significant throughput benefits.

### 2.2.2. Message Stability

Publishing messages to a queue is a coordinated hand-off between the publishing application and the queue. The queue assumes responsibility for delivering a message only when the publisher is informed that the message is *stable*. Stability refers to the queue having a copy of the message in memory and/or on disk, depending on configuration. Until the publisher is informed that a message is stable, it may be lost upon restart. Messages may be delivered without the publisher being informed that they were stable. Upon restart, the publisher is expected to send again any messages previously sent that were not known to be stable. Since the queue cannot differentiate between new messages and messages sent again upon publisher restart, the application is responsible for marking messages as sent again in some manner (e.g., by setting a flag in message content or properties), if required for downstream duplicate checking.

### 2.2.3. Once-and-Only-Once Delivery

Once-and-Only-Once (OAOO) delivery means that each message is assigned to only one receiver at a time. If a message is not acknowledged by the assigned receiver, it can be reassigned and redelivered to either the same or a different receiver (depending on configuration and circumstances), in which case the resent message is flagged as redelivered. Applications are always responsible for detection and proper handling of potentially duplicate messages.

### 2.2.4. Application Sets

An Application Set is a group of receivers and can be used to load balance queue topics within a receiving application or accommodate multiple processing purposes for a single topic. The OAOO semantic applies to an

Application Set. Therefore, you can configure multiple Application Sets for a queue and only one receiver in each set will process a given message. See *Application Set Element*.

### 2.2.5. Receiver Portion Size

You can increase the throughput to receivers by increasing their portion size. This increases the number of messages in flight to a receiver. This setting is specified in a Receiver Type ID. See *Options for a Receiver Type's ume-attributes Element*. A receiver configured with one Receiver Type ID and subscribed to Topic A in one application can have 5 messages in flight, where a Topic A receiver configured with a different Receiver Type ID in another application can have only one message in flight.

### 2.2.6. Configurable Store and Forward

You can configure a Queue to assign and send data to receivers after the data has been persisted to disk (**Store-Then-Forward**) or in parallel to being persisted to disk (**Store-While-Forwarding**). **Store-While-Forwarding** produces lower end-to-end latency from sources to receivers at the expense of potential message loss under certain multiple failure conditions. See also ...

- *Queue Element*
- *Queuing Architecture*

### 2.2.7. Multiple Dissemination Models

**UMQ** provides the following load balancing configurations for data dissemination not possible with other queuing products.

- *Serial Queue Dissemination (SQD)*: Queue sends data to each receiver via serial unicasts. Only the receivers assigned to a message receive that message.
- *Parallel Queue Dissemination (PQD)*: Queue sends data and control information to all receivers via **UM** transport sessions. Control information contains assignment information.
- *Source Dissemination (SD)*: Source sends data to all receivers via **UM** transport sessions. Queues send control information to all receivers via separate transport sessions. Control information contains assignment information.

### 2.2.8. Queue Fault Tolerance

Queues may be composed of several actual queue instances that operate with source and receivers in such a way that failure of one queue instance or even several queue instances need not stop or even slow down sources and receivers from performing queuing operations. See *Queue Redundancy*.

Known Issue: Configurations with multiple queue instances (slaves) can lead to inconsistent state, which can trigger message loss, crashes or restart issues necessitating removal of files resulting in message loss. Therefore, Informatica recommends deploying configurations with only a single queue instance (without slaves). To facilitate failover, set the `sinc-log-filename`, `sinc-data-filename`, and `sinc-queue-swap-filename` to write to a shared file-system, and use external process management (automatic or manual) to start up a secondary queue instance referencing the same files if and only if the active instance fails (i.e. only allow one queue instance to access the files

at any time). With this configuration, sinc files will grow over time, so clean restarts (i.e. shut down, delete all files and restart) will be required periodically. Using a shared file-system may impact performance; Informatica strongly recommends holistic system performance characterization prior to any production deployment.

### 2.2.9. Indexed Queuing

Messages may be sent with an *index* using an extended send call, `lbm_src_sendv_ex`, that includes a pointer to `lbm_umq_index_info_t` (../API/structlbm__umq__index__info__t__stct.html) in `lbm_src_send_ex_info_t`. An index is an application-defined 64-bit unsigned number or free-form string.

By default, all receivers are eligible to be assigned indices by the Queue. Once the Queue assigns the first message sent with a particular index to an individual receiver in each Application Set, the Queue assigns subsequent messages (sent from any source) with that same index to those same individual receivers, provided the receivers remain alive and responsive.

You can exert greater control over how a Queue assigns indices to receivers by configuring individual indices and ranges of indices in the Queue's `umestored` XML configuration file. You can then set rules that allow or deny receivers the permission to process messages with certain indices. See *Indices Element*.

Note that with indexed queuing, messages with a particular index can be assigned only to the single consumer responsible for that index (i.e., assignment is "sticky"). If an index consumer stops consuming messages (a failure scenario) long enough for its configured portion size to become full while another message with the same index is pending, message delivery to all other consumers within the same Application Set is halted. Therefore, Informatica recommends using indexed queuing only in configurations with message reassignment enabled and set to a relatively brief interval, as this determines the amount of time that message consumption within an Application Set may be halted in the event of receiver failure.

### 2.2.10. Dead Letter Queue

Queues can be configured to isolate unconsumed messages in a Dead Letter Queue, which prevents these messages from causing application or queuing system problems. These unconsumed messages remain in the Dead Letter Queue for the life of the Queue. Other applications can access these messages for analysis by starting a wildcard receiver for the Dead Letter topic queue.

Configuring a Dead Letter Queue involves the following actions in the Queue's `umestored` XML configuration file.

1. Configure a `dead-letter-topic-prefix` and `dead-letter-topic-separator` for the Queue. **UMQ** uses this information to compose a Dead Letter topic name. The example below illustrates this step, but may or may not be suitable to include in your `umestored` XML configuration file. See also *General Options for a Queue's ume-attributes Element*.

   ```
   <queues>
   <queue name="Sample_Queue_with_Dead_Letter_Topic" port="20333" group-index="0">
   <ume-attributes>
   <!-- dead-letter-topic-prefix must be specified for any dead letter queues to be created -->
   <option type="queue" name="dead-letter-topic-prefix" value="dead_letter"/>
   <!-- dead-letter-topic-separator defaults to '/', so the following line isn't necessary -->
   <option type="queue" name="dead-letter-topic-separator" value="/"/>
   </ume-attributes>
   </queue>
   ```

2. In each of the Queue's Application Sets from which you wish to capture unconsumed messages, set the `discard-behavior` to `dead-letter`. The example below illustrates this step, but may or may not be suitable to include in your `umestored` XML configuration file. See also *Options for an Application Set's ume-attributes Element*.

```
<application-sets>
<application-set name="Set 1">
<ume-attributes>
<option type="queue" name="log-audit-trail" value="1"/>
<!-- discard-behavior set to dead-letter tells the queue that when messages
EOL off this appset, they should be placed on a dead letter queue -->
<option type="queue" name="discard-behavior" value="dead-letter"/>
</ume-attributes>
</application-set>
</application-sets>
```

3. Configure a `dead-letter-topic` for the Queue and assign it to a different `application set` than the one in step 2 that has its `discard-behavior` set to `drop`. The example below illustrates this step, but may or may not be suitable to include in your `umestored` XML configuration file. See also *Options for a Queue Topic's ume-attributes Element*.

```
<topics>
<!-- since the dead-letter-topic-prefix is dead_letter, it is necessary to specify a topic
pattern that matches dead_letter -->
<topic pattern="dead_letter" type="PCRE">
<ume-attributes>
<!-- this topic is a dead-letter-topic -->
<!-- you cannot create a source and submit messages directly to a dead
letter topic -->
<option type="queue" name="dead-letter-topic" value="1"/>
</ume-attributes>
<application-sets>
<!-- you must assign this dead-letter-topic to an application set that has its
"discard-behavior" set to "drop" -->
<application-set name="Set 2"/>
</application-sets>
</topic>
</topics>
```

## 2.2.11. Message Lifetimes

You can configure a lifetime period for messages during which the message may be assigned to a receiver. Upon expiration of the message lifetime, the queue cannot assign the message to a receiver. The queue either discards the message from the queue permanently or sends it to the *Dead Letter Queue*, if configured. A message's total lifetime starts when the queue enqueues the message. Messages lifetimes apply to both **UMQ** and ULB messages. See *Message Lifetimes and Reassignment*.

## 2.2.12. Queuing Terminology

**Ultra Messaging Queuing Edition** uses the following terms.

| Term | Description |
|---|---|
| Queue | A named, virtual entity to which sources submit messages and from which receivers retrieve messages. |
| Queue Instance | A running daemon that is part of a Queue. In **UMP** , this is the `umestored` process. The collection of all queue instances with the same name constitutes a Queue. |
| Registration ID | The ID that a source or receiver application uses to register a context with a Queue. The context uses the same Registration ID for each Queue Instance. The context may generate a Registration ID or the user may specify Registration IDs to be used with specific Queues. See `umq_queue_registration_id` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQQUEUEREGISTRATIONID) |
| Message ID | The unique ID of a queue message. |
| Assignment | The designation of particular queued messages to individual receivers for processing. |
| Consumption | The processing of a queue message by an assigned receiver. |
| Consumption Report (CR) | Receivers send Consumption Report (CR) messages to notify Queues of message consumption. |
| Re-Assignment | When a receiver does not consume a message within a specified period of time, the Queue re-assigns the message to another receiver. |
| Assignment ID | Used by the Queue to identify receivers registered with certain topics. Found within control information. |
| Receiver Control Record (RCR) | Message control information sent to receivers by Queues using Parallel Queue Dissemination (PQD) or Source Dissemination (SD). An RCR contains the Message ID of the message, a list of the Assignment IDs (receivers) that should process the message, and ordering information. |
| RCR Index | An index that identifies a topic on a Queue. |
| End of Lifetime (EOL) | Maximum time limit before a message must be consumed. The timer starts when the Queue assigns the message to a receiver and is unaffected by reassignments. The Queue discards any message with an expired EOL. |
| Receiver Type ID | Indicates the Application Set a receiving application wishes to join. This ID is fully managed by the administrator of the Queue and implies not only Application Set but also some other ways that the receiver is treated, such as portion size and assignment options. |

# 3. Architectures

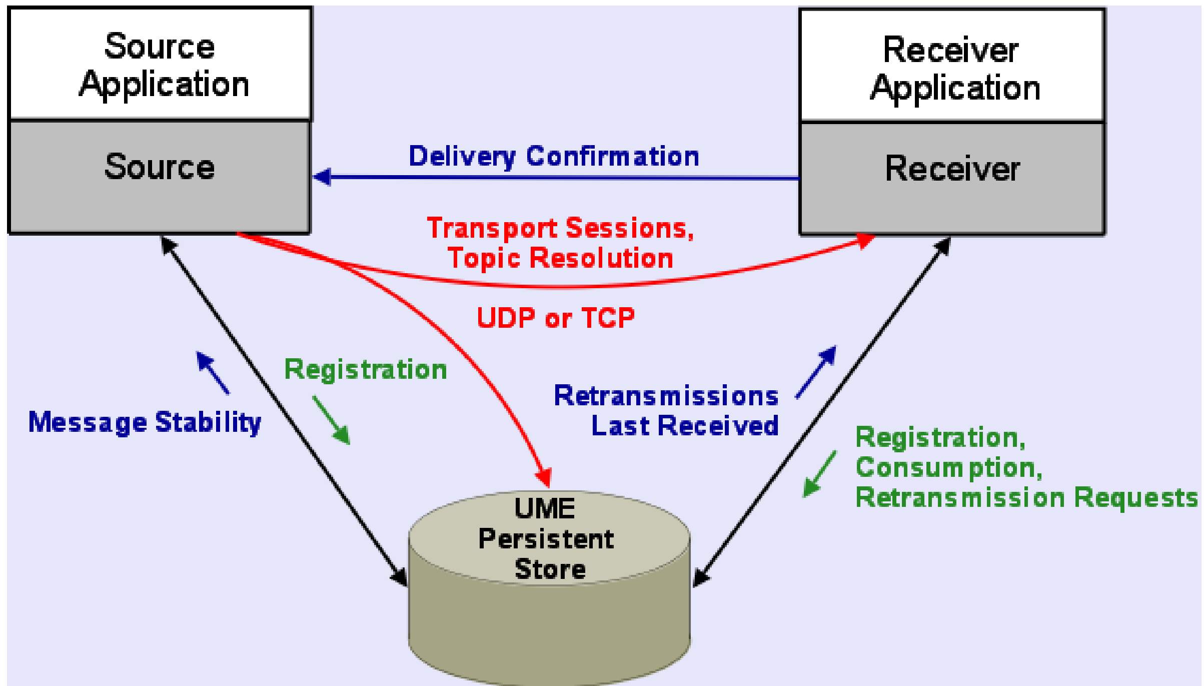
The same **Ultra Messaging** API may be used for stream-based messaging or persistent messaging and queuing . Similarly, the `umestored` daemon can be configured as a persistent store or queue, providing consistent and efficient operation across persistent and queuing messaging systems.

## 3.1. Persistence Architecture

As shown in the diagram, **UMP** provides messaging functionality as well as persistent operation. See *UMP*

*Persistence Architecture* for an overview of **UMP** architecture.

**Figure 1. UMP Persistence Architecture**




The highlights of this architecture are:

- Sources communicate with stores
- Receivers communicate with stores
- Sources communicate with receivers

> **Note:** The persistent store does not lie in the middle of the data path between source and receivers. Along with other enhancements, this feature, called **Parallel Persistence**, gives **UMP** a significant performance edge over any other persistent messaging product.

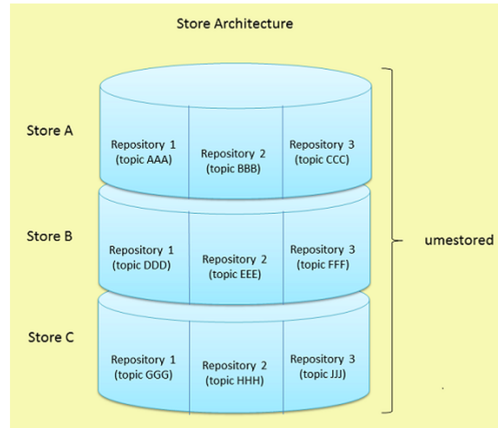
> **Note:** The persistent store is not supported on the HP NonStop® platform.

## 3.2. Persistent Store Architecture

The `umestored` daemon runs the **UMP** persistent store feature. You can configure multiple stores per daemon using the `<store>` element in the `umestored` XML configuration file. See *Configuration Reference for Umestored*.

Individual stores can use separate disk cache and disk state directories and be configured to persist messages for multiple sources (topics), which are referred to as, source repositories. **UMP** provides each `umestored` daemon with a Web Monitor for statistics monitoring. See *Ultra Messaging Web Monitor*.

**Figure 2. Store Architecture**




This section discusses the following topics.

- *Source Repositories*
- *Persistent Store Fault Tolerance*

## 3.2.1. Source Repositories

Within a store, you configure repositories for individual topics and each can have their own s et of `<topic>` level options that affect the repository's type, size, liveness behavior and much more. If you have multiple sources sending on the same topic, **UMP** creates a separate repository for each source. **UMP** uses the repository options configured for the topic to apply to each source's repository. If you specify 48MB for the size of the repository and have 10 sources sending on the topic, the persistent store requires 480MB of storage for that topic.

A repository can be configured as one of the following types.

- `no cache` - the repository does not retain any data, only state information
- `memory` - the repository maintain both state and data only in memory
- `disk` - the repository maintains state and data on disk, but also uses a memory cache.
- `reduced-fd` - the repository maintains state and data on disk, also uses a memory cache but uses significantly fewer File Descriptors. Normally a store uses two File Descriptors per topic in addition to normal UM file descriptors for transports and other objects. The reduced-fd repository type uses 5 File Descriptors for the entire store, regardless of the number of topics, in addition to normal UM file descriptors for transports and other objects. Use of this repository type may impact performance.

You can configure any combination of repository types within a single store configuration.

**Note:** If you run a store with all `disk` or `reduced-fd`type repositories, then restart the store with memory type repositories and do not clear out the `disk-cache-directory` and `disk-state-directory`, the memory repositories revert automatically to disk repositories.

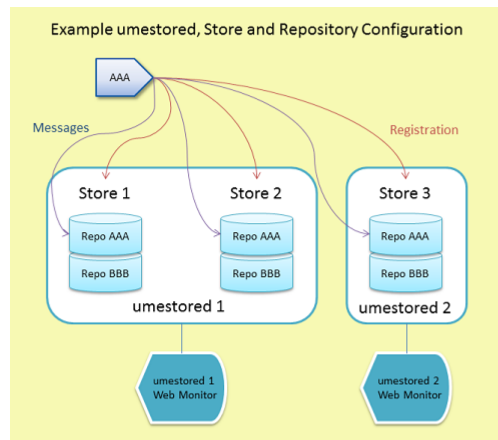
**Note:** With **UMP** Version 5.3, the **UMP** store daemon has Standard C++ Library dependencies for Unix packages. The `libstdc++` must also be included in LD_LIBRARY_PATH. See Section 3. Code (../../DocIntro.html#CODE) for more infromation.

### 3.2.2. Persistent Store Fault Tolerance

Sources and receivers register with a store and use individual repositories within the store. Sources can use redundant repositories configured in multiple stores in either a Round Robin or Quorum/Consensus arrangement for fault tolerance. Stores and repositories have no indication of these arrangements.

The following diagram depicts an example Quorum/Consensus configuration of stores and repositories. These stores could also be run by a single `umestored` daemon or one daemon for each store.

**Figure 3. Example Store Configuration**




See *Store Configuration Considerations* and also *Stores Element* for more about store configuration.

## 3.3. Queuing Architecture

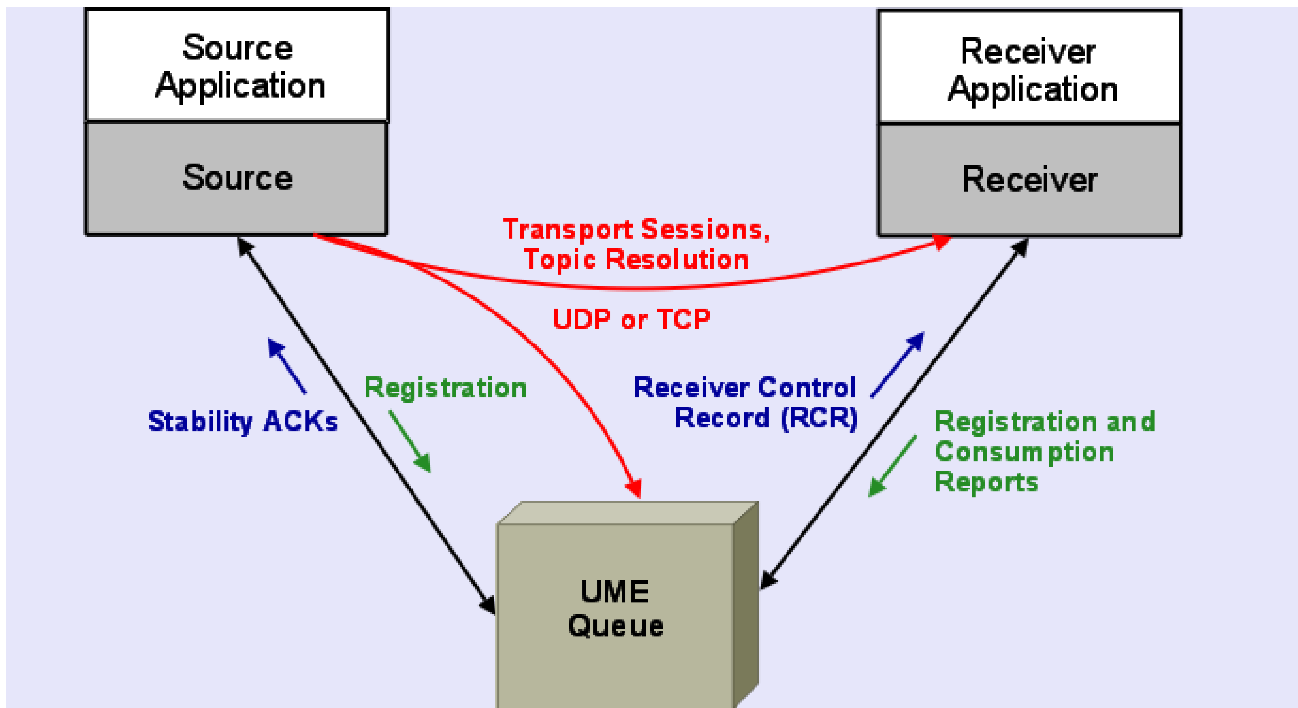
The architecture of Queues follows a lot of the same tenets as the **UMP** persistence architecture. Source and receiver applications can create sources, listen on topics and do normal operations typical of **UM** applications. Receivers require no special configuration to receive messages from a queue.

The central components to any queuing deployment are:

- the source applications
- the receiving applications
- the `umestored` daemon, which provides queue instances

The `umestored` daemon provides separate Queue instances. Just as `umestored` may contain individual **UMP** stores, it may also contain individual queue instances as well.

**Figure 4. Queuing Architecture**




# 4. Operational View

This section discusses the following topics.

- *Persistence Operations*
- *Receiver-paced Persistence Operations*
- *Queuing Operations*
- *Ultra Load Balancing Operations*
- *UMP and UMQ Events*

> **Note:** If your application is running with the **UM** configuration option, `request_tcp_bind_request_port` (../Config/requestnetworkoptions.html#CONTEXTREQUESTTCPBINDREQUESTPORT) set to zero, request port binding has been turned off, which also disables **UMP** .

## 4.1. Persistence Operations

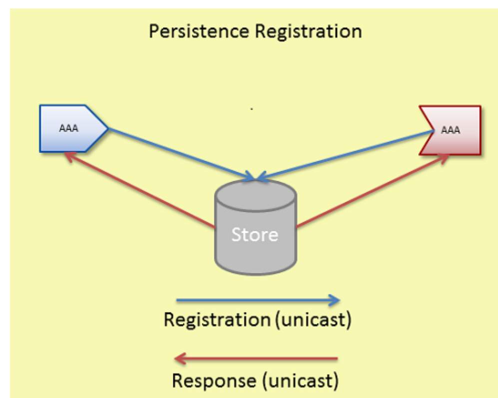
Sources, receivers, and stores in **UMP** interact in very controlled ways. This section illustrates the flow of network traffic between the components during three modes of operation and also provides a reference of **UMP** Events.

- *Registration*
- *Normal Operation*
- *Receiver Recovery*

### 4.1.1. Registration

Figure 5 illustrates network flow during the Registration process.

**Figure 5. UMP Registration**

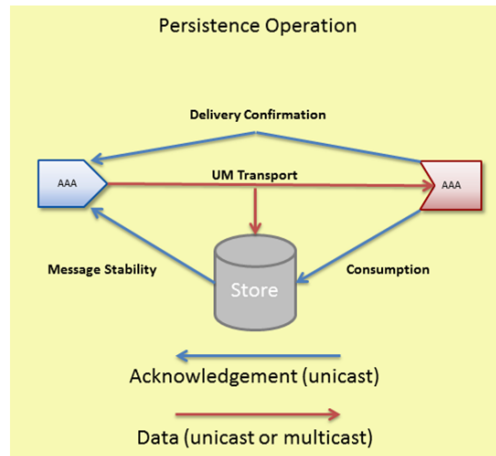



Sources and receivers unicast registrations to the store. The store unicasts responses back to the sources and receivers. Registrations are on a per topic per source basis. Stores use RegIDs to identify sources and receivers. After registration, receivers may handle recovery, sources may send data, and receivers may send acknowledgements

### 4.1.2. Normal Operation

Figure 6 illustrates the normal operation of data reception and acknowledgement and also shows how **UMP** attains **Parallel Persistence**. The source sends message data to receivers and stores in parallel.

**Figure 6. UMP Normal Operation**




1. Sources transmit data to receivers and stores at the same time over **UM** multicast or unicast transport protocols.
2. As the store receives and persists messages, the store unicasts acknowledgements, (message stability control messages), to the source letting it know of successful reception and storage.
3. As receivers process and consume messages they unicast acknowledgments to the store letting the store know of successful consumption of data.
4. If the source desires delivery confirmation, the receiver unicasts acknowledgements directly to the source letting the source know of message consumption as well.

Normal operation and recovery can proceed at the same time. In addition, as a receiver consumes retransmitted messages, the receiver sends normal acknowledgements for consumption and confirmed delivery (if requested by the source).

> **Note:** A store can be configured with different storage limits for each repository. If the repository reaches this limit, the repository releases the oldest message in order to persist a new message. This behavior occurs for a memory repository as well as a disk repository. If a repository releases a message that one or more receivers have not consumed (sent a consumption notification), the repository logs a single warning message in the store log file per receiver per registration.

#### *4.1.2.1. **UMP** Flight Size*

**UMP** supports a flight size mechanism that tracks messages in flight from a particular source and responds when a send would exceed the configured flight size ( `ume_flight_size` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZE) and/or `ume_flight_size_bytes` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZEBYTES)). You can configure `ume_flight_size_behavior` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZEBEHAVIOR) to either:

- block any sends that would exceed the flight size or,
- allow the sends while notifying your application.

**UMP** considers a sent message in flight until the following two conditions are met.

1. The source receives the configured number of stability acknowledgements from the store(s).
2. The source has received the configured number of delivery confirmation notifications. (See `ume_retention_unique_confirmations` (../Config/config.html#SOURCEUMERETENTIONUNIQUECONFIRMATIONS).)

If configuring both `ume_flight_size` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZE) and `ume_flight_size_behavior` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZEBEHAVIOR), **UMP** uses the smaller of the two flight sizes on a per send basis.

| ume_flight_size | ume_flight_size_bytes | Result |
|---|---|---|
| Exceeded | Exceeded | ume_flight_size_behavior executes |
| Exceeded | Not Exceeded | ume_flight_size_behavior executes |
| Not Exceeded | Exceeded | ume_flight_size_behavior executes |
| Not Exceeded | Not Exceeded | No flight size sending restriction |

When using stores in a Quorum/Consensus configuration, intragroup (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERETENTIONINTRAGROUPSTABILITYBEHAVIOR) and intergroup (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERETENTIONINTERGROUPSTABILITYBEHAVIOR) stability settings affect whether **UMP** considers a messages in flight. Consider a case with three stores in a single QC group, and two receivers. Given the default configuration, until a source receives a stability notification from two of the three stores, **UMP** considers a given message in-flight. In addition, if you set `ume_retention_unique_confirmations` (../Config/config.html#SOURCEUMERETENTIONUNIQUECONFIRMATIONS) to 2, that same message would be considered in flight until the source receives two stability notifications AND two delivery confirmation notifications. See also *Sources Using Quorum/Consensus Store Configuration*.

> **Note:** The **UMP** flight size mechanism operates on a per message basis, not a per fragment basis.

> **Note:** The **UMP** flight size bytes mechanism operates with only payload data. **UM** or network overhead is not included in the byte count.

#### *4.1.2.1.1. Blocking Message Sends That Exceed the Flight Size*

By default, when a source sends a message that exceeds it's flight size, the call to send blocks. For example, suppose the flight size is set to 1. The first send completes but before the source receives a stability notification or delivery

confirmation, it initiates a second call to send. If the source uses a blocking send, the send call blocks until the first message stabilizes. If the source uses a non-blocking send, the send returns an LBM_EWOULD_BLOCK.

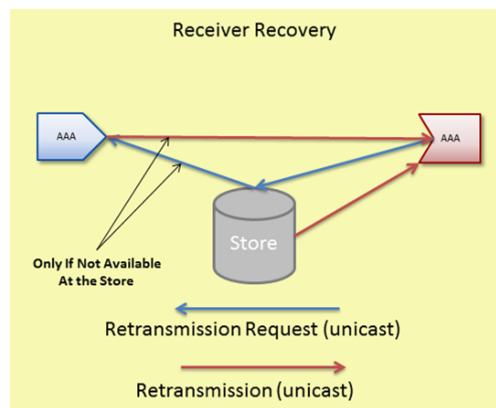
#### *4.1.2.1.2. Notification of Message Sends That Exceed the Flight Size*

Alternatively, `ume_flight_size_behavior` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZEBEHAVIOR) can be set to notify your application when a message send surpasses the flight size. A send that exceeds the configured flight size succeeds and also triggers a flight size notification, indicating that the flight size has been surpassed. Once the number of in-flight messages falls below the configured flight size, another flight size notification source event is triggered, this time, informing the application that the number of in-flight messages is below the source's flight size.

## 4.1.3. Receiver Recovery

Figure 7 illustrates receiver recovery.

**Figure 7. UMP Messages Recovery**




Receivers unicast retransmission requests. If the store has the message, it unicasts the retransmission to the receiver. If it does not have the message and is configured to forward the request to the source (See `retransmission-request-forwarding` in *Options for a Topic's ume-attributes Element*), it unicasts the retransmission request to the source. If the source has the message, it unicasts the retransmission directly to the receiver.

**UM** sends retransmissions from a thread separate from the main context thread so as not to impede live message data processing. The `<store>` configuration option, `retransmission-request-processing-rate`, sets the store's capacity to process retransmission requests. The retransmission thread processes requests off a retransmission queue which is set at 4 times the size of `retransmission-request-processing-rate`. The following **UM** Web Monitor statistics record retransmission activity. See **UM** Web Monitor *Store Page*.

- Retransmission requests received rate

- Retransmission requests served rate
- Retransmission requests dropped rate
- Total retransmission requests dropped since store startup

## 4.2. Receiver-paced Persistence Operations

Receiver-paced Persistence (RPP) refers to different message retention behavior for designated receivers. You enable RPP with **UM** configuration options. No special API calls are needed. RPP differs from **UMP**'s default source-paced persistence in the following ways.

- The repository must be configured to allow RPP and sources and receivers must be configured to request RPP behavior during registration.
- Sources can modify specific repository configuration options that pertain to RPP.
- The repository retains RPP messages until all RPP receivers acknowledge consumption. The repository maintains an accurate count of all RPP receivers.
- Late Joining receivers cannot receive all previously sent topic messages, only those unconsumed by all RPP receivers. Late Joining receivers can always start at the current message retained by the repository, defined as the earliest message not consumed by all RPP receivers.
- Sources must also configure their flight size in bytes, and optionally, in message count. By using a total bytes flight size, the store can keep track of exactly how must space it has available and not send stability acknowledgements if new messages would exceed the available space, which would endanger the receipt of all messages by all RPP receivers. See *UMP Flight Size*.

In addition, a disk write delay interval for the repository, available for Source-paced Persistence as well, improves performance by preventing unnecessary disk activity.

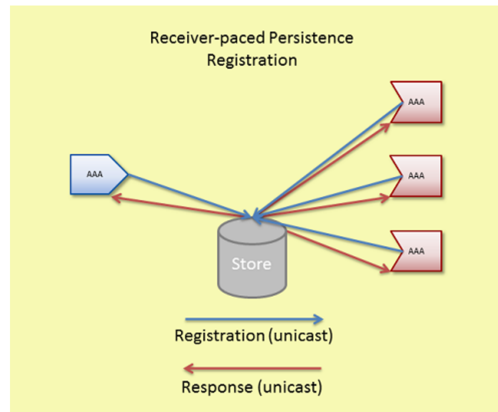
This section discusses the following topics.

- *RPP Registration*
- *RPP Normal Operations*
- *RPP Message Recovery*
- *RPP Deregistration*
- *Implementing RPP*
- *Example RPP Configuration Files*
- *RPP Cross Feature Functionality*

### 4.2.1. RPP Registration

If a source sets ume_receiver_paced_persistence (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERECEIVERPACEDPERSISTENCE), its topic becomes a RPP topic. When the source registers with the store, the source's repository also becomes a RPP repository. Receivers registering with a store on the RPP topic become RPP receivers.

**Figure 8. RPP Registration**




A source registration request includes the following.

- Designation of a RPP topic (LBMC_UME_PREG_FLAG_REGISTER flag)
- Reconfigured repository configuration option values. Possible options are the 3 repository size options, `repository-allow-ack-on-reception`, `repository-disk-write-delay` and `source-flight-size-bytes-maximum`.
- Re-registration must request same configuration options or the store rejects the request.

Receiver registration request includes its designation as a RPP receiver (LBMC_UME_PREG_FLAG_REGISTER flag).

The repository's registration response to both a source and a receiver acknowledges RPP mode.

#### *4.2.1.1. Late Registering Receiver*

Late joining receivers that register after the first RPP topic message has been sent cannot receive any messages sent prior to their registration, except for messages not yet consumed by all RPP receivers. This behavior also applies to the very first receiver of a RPP group that registers after the source sends the first message. Any messages published prior to RPP receiver registration are not available for recovery.

#### *4.2.1.2. Early Exiting Receiver*

Should a registered receiver's activity timer expire and be declared by the repository to be inactive, the repository retains all messages published since the receiver's last acknowledged message (or initial sequence number if no messages were acknowledged) until its receiver state lifetime expires and the repository deletes the receiver state information. Deleting receiver state removes all knowledge of the receiver from the repository. As a result, the repository also deletes all messages being held solely for this receiver.

Should an early exiting receiver reregister (or otherwise become active) before the expiration of its state lifetime, that receiver can recover all messages retained for that receiver.

#### 4.2.1.3. UMP Version RPP Compatibility Matrix

The following table indicates the result of registration requests across **UMP** versions.

| Version/Object | Pre-ver. 5.3 Store | Ver. 5.3 RPP Store | Ver. 5.3 Non-RPP Store |
|---|---|---|---|
| Pre 5.3 Source | Granted | Rejected * | Granted * |
| 5.3 RPP Source | Granted - Source Error | Granted * | Rejected * |
| 5.3 Non-RPP Source | Granted | Rejected * | Granted * |
| | | | |
| Pre 5.3 Receiver | Granted | Rejected | Granted |
| 5.3 RPP Receiver | Granted - Receiver Error | Granted | Rejected |
| 5.3 Non-RPP Receiver | Granted | Rejected | Granted |

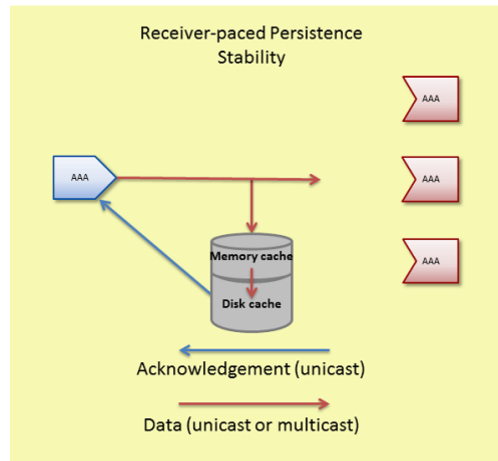
- **Granted - Source Error** indicates that the store granted the registration but the source detected that RPP behavior was not acknowledged by the store.
- **Granted - Receiver Error** indicates that the store granted the registration but the receiver detected that RPP behavior was not acknowledged by the store.
- * Refers only to the re-registration of a source with an existing source repository because the source determines the repository's behavior for new registrations.

### 4.2.2. RPP Normal Operations

Since all RPP receivers must receive all messages, message overruns at the store or receiver must be prevented by regulating the sending pace of the source. The store uses the source's flight size (bytes) to regulate the source's speed, by withholding stability acknowledgements if the repository does not have at least one flight size available.

1. Sources transmit data to receivers and store repositories at the same time over **UM** multicast or unicast transport protocols.
2. When a disk repository receives a message, it holds the message in memory cache before it writes the message to disk. The repository sends a stability notification to the source after it writes the message to disk. Memory repositories send the stability notice upon reception. See also Acknowledge on Reception and Receiver Acknowledgement and Flight Size below.

**Figure 9. RPP Stability Acknowledgement**

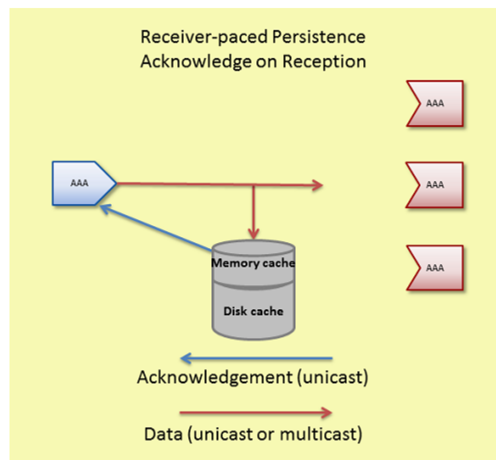



3. If the source desires delivery confirmation, receivers unicast acknowledgements directly to the source letting the source know of message consumption as well.

The following also affect when a repository sends a stability acknowledgement to the source.

- **Acknowledge on Reception** - If you configure the repository for `repository-allow-ack-on-reception` and the source also sets `ume_repository_ack_on_reception` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYACKONRECEPTION), the repository sends a stability acknowledgement to the source immediately upon reception. If the disk write has not already been initiated, **UMP** does not write the message to disk.

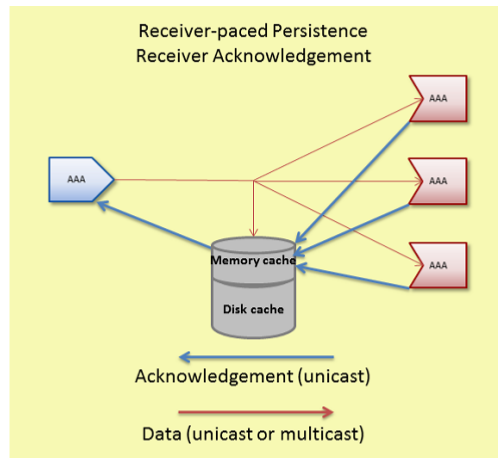
**Figure 10. RPP Acknowledge on Reception**




- **Receiver Acknowledgement** - If a repository receives acknowledgements from all receivers before writing the message to disk, it immediately sends a stability acknowledgement to the source. If the disk write has not already

been initiated, **UMP** does not write the message to disk.

**Figure 11. RPP Receiver Acknowledgement**




- **Write Delay** - The repository option, repository-disk-write-delay, allows the repository to hold messages in memory cache longer before persisting them to disk. This delay increases the probability that all RPP receivers acknowledge message consumption, eliminating the need to persist the message to disk.
- **Flight size** - A disk repository only sends stability acknowledgement to the source if its memory cache has at least one flight size (in both messages and bytes) available. A memory repository also sends stability acknowledgement if it has at least one flight size (in both messages and bytes) available. A lack of available space in the repository blocks the source until the repository reclaims the necessary storage space and sends a stability acknowledgement.

For memory store repositories, the behaviors Acknowledge on Reception, Receiver Acknowledgement and Write Delay do not apply.

## 4.2.3. RPP Message Recovery

An RPP source repository retains messages until all RPP receivers acknowledge receipt of the message. Therefore an RPP receiver can only recover messages that have not been consumed by all RPP receivers. It is important to note that an RPP receiver joining after other RPP receivers have already joined and after messages have already been sent can only be guaranteed to recover messages sent subsequent to its joining.

## 4.2.4. RPP Deregistration

You can deregister either sources or receivers using deregistration APIs, (`lbm_src_ume_deregistration()`, `lbm_rcv_ume_deregistration()` and `lbm_wrcv_ume_deregistration()`). **UM** deletes the state of deregistered objects. If you deregister a RPP receiver, **UMP** automatically updates the number of receiver acknowledgements required to maintain RPP behavior. The store issues Deregistration Successful events for every source or receiver that deregisters. See *UMP and UMQ Events*.

Applications should be cautious about using the deregistration APIs to deregister RPP sources or receivers. These APIs can be disruptive to RPP.

- `lbm_src_ume_deregistration()` also deletes any persisted RPP messages in the source's repository. A source application should only use `lbm_src_ume_deregistration()` if it uses delivery confirmation from the receiver and it knows all messages have been delivered. The source is blocked after deregistering and and must restart in order to register again with the RPP store.
- Deregistering an RPP receiver with `lbm_rcv_ume_deregistration()` removes the receiver from the list of RPP receivers maintained by the repository. It is no longer part of the persistence operation, but is a valid **UM** receiver, able to receive messages, but unable to acknowledge message consumption to the repository. Any messages not yet confirmed for that receiver are unrecoverable. The receiver must restart in order to register again with the RPP repository.
- Deregistering an RPP wildcard receiver with `lbm_wrcv_ume_deregistration()` deregisters all individual topic receivers receiving messages on topics that match the wildcard pattern. Individual topic receivers can still receive messages after deregistering, but cannot acknowledge message consumption. The wildcard receiver must restart in order to register again with the RPP store.

### 4.2.5. Implementing RPP

Follow the procedure below to configure Receiver-paced Persistence.

1. Set `ume_receiver_paced_persistence` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERECEIVERPACEDPERSISTENCE) for sources and receivers in a **UM** configuration file. If only certain sources or receivers in a context are RPP, use `lbm_*setopt()`l in the source or receiver application or use **Ultra Messaging Manager** to specify RPP in an **UM** XML configuration file.
2. Set `repository-allow-receiver-paced-persistence` = 1 for the repository in the `umestored` XML configuration file.
3. Coordinate `ume_flight_size_bytes` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZEBYTES) between the repository and the source. Set the maximum flight size with the repository option, `source-flight-size-bytes-maximum`. Sources can reconfigure the repository's `source-flight-size-bytes-maximum` to a value less than or equal to the maximum.
4. **Optional.** Coordinate the `ume_repository_ack_on_reception` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYACKONRECEPTION) between the repository and the source. If the repository has `repository-allow-ack-on-reception` enabled (1), the source can choose to keep it enabled or turn it off ( `ume_repository_ack_on_reception` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYACKONRECEPTION) = 0). If the repository has `repository-allow-ack-on-reception` disabled (0), the source cannot turn it on.
5. **Optional.** If the repository is a disk repository (repository-type = `disk` or `reduced-fd`), set the maximum write delay with the repository option, `repository-disk-write-delay`. Sources can reconfigure the repository's `repository-disk-write-delay` to a value less than or equal to the maximum configured for the repository with `ume_write_delay` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEWRITEDELAY).
6. **Optional.** Coordinate repository size options between the source and repository. If you wish to use the repository's values, you do not need to configure source configuration values. The repository sets a maximum for these three options. The source can reconfigure the repository's options with values less than or equal to the maximum configured for the repository using the following **UM** configuration options.

- `ume_repository_size_threshold` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYSIZETHRESHOLD)
- `ume_repository_size_limit` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYSIZELIMIT)
- `ume_repository_disk_file_size_limit` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYDISKFILESIZELIMIT)

## 4.2.6. Example RPP Configuration Files

The sample configuration files shown below show how a store configuration file establishes certain RPP option values and the source can reconfigure them via a **UM** configuration file. Although only two files appear below, this configuration represents two, single-store quorum/consensus groups and one **UM** context. A second `umestored` configuration file would be required for the store `store1rpp` containing options and values identical to `store0rpp`.

### *4.2.6.1. UM Configuration File*

The following example **UM** configuration file contains RPP options in the `##Persistence Options###` section.

- The source uses the same repository size values as the store. In this case, you do not need to specify these option values again in the source's **UM** Configuration File. They appear in this file for the sake of completeness.
- The source reconfigures ume_flight_size_bytes to 1,000,000 bytes, which is less than the repository's 4 MB default. (The source can reconfigure this option to a value less than or equal to the repository's configured value.)
- The source reconfigures ume_write_delay from the default of 0 ms to 1000 ms or 1 second.
- The option, ume_session_id 5353, is commented out because this file specifies RegIDs 2929 and 2930, respectively, for the stores in the `ume_store_name` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTORENAME) option. The option, `ume_session_id` (../Config/ultramessagingpersistenceoptions.html#CONTEXTUMESESSIONID), appears in this file as a reminder that you can use either RegIDs or Session IDs, but not both.

```
#Sample UM Configuration File, UMP Version 5.3
#Major Options
source transport lbtrm
# in order and reassembled
receiver ordered_delivery 1
#Multicast Resolver Network Options
context resolver_multicast_address 225.8.17.29
context resolver_multicast_interface 10.29.3.0/24
# Transport LBT-RM Netowrk Options
source transport_lbtrm_multicast_address 225.8.17.30
context transport_lbtrm_multicast_address_low 225.12.17.10
context transport_lbtrm_multicast_address_high 225.12.17.14
#Transport LBT-RM Operation Options
context transport_lbtrm_data_rate_limit 10000000
context transport_lbtrm_retransmit_rate_limit 5000000
# Transport LBT-RM Reliability Options
receiver transport_lbtrm_nak_initial_backoff_interval 40000
```

```
receiver transport_lbtrm_nak_initial_backoff_interval 500
receiver transport_lbtrm_nak_generation_interval 10000
##Turn off NAKs
receiver transport_lbtrm_send_naks 0
#Request Network Options
context request_tcp_port_low 55000
context request_tcp_port_high 55500

## Persistence Options ###
source ume_store_group 0:1
source ume_store_name store0rpp:2929:0
source ume_store_group 1:1
source ume_store_name store1rpp:2930:1
source ume_store_behavior qc
source ume_flight_size 500
source ume_flight_size_bytes 1000000
source ume_receiver-paced-persistence 1
source ume_repository_size_threshold 104857600
source ume_repository_size_limit 209715200
source ume_repository_disk_file_size_limit 1073741824
source ume_repository_ack_on_reception 1
source ume_write_delay 1000
receiver ume_receiver-paced-persistence 1
receiver ume_explicit_ack_only 1
source ume_proxy_source 1
#source ume_session_id 535353
context ume_source_liveness_timeout 4000
context ume_receiver_liveness_interval 1000
source ume_confirmed_delivery_notification 1
```

#### *4.2.6.2. umestored Configuration File*

The following example store configuration file contains RPP options in the ABC* topic section.

- The store has raised the `repository-size-limit` from the default of 48 MB to 200 MB, the `repository-size-threshold` from the default of 0 to 100 MB, and the `repository-disk-file-size-limit` from the default of 100MB to 1 GB.
- The store does not specify a `source-flight-size-bytes-maximum`, using the default of 4 MB.

```
<?xml version="1.0"?>
<ume-store version="1.2">
  <daemon>
    <log>/configs/stores/umestored1/umestored.log</log>
    <lbm-license-file>/bin/umq_exp_license.txt</lbm-license-file>
    <lbm-license-file>/bin/lbm_ume_umq_udx_rdma_license.txt</lbm-license-file>
    <lbm-config>/configs/lbm_4_store.cfg</lbm-config>
    <pidfile>/configs/stores/umestored1/umestored.pid</pidfile>
    <web-monitor>*:15404</web-monitor>
  </daemon>
  <stores>
    <store name="rpp-ump-test-store-1" port="14667">
```

```
      <ume-attributes>
        <option type="store" name="disk-cache-directory" value="/stores/store1/cache"/>
        <option type="store" name="disk-state-directory" value="/stores/store1/state"/>
        <option type="store" name="allow-proxy-source" value="0" />
        <option type="store" name="context-name" value="store1rpp"/>
      </ume-attributes>
      <topics>
        <topic pattern="ABC*" type="PCRE">
          <ume-attributes>
             <option type="store" name="repository-allow-receiver-paced-persistence" value="1"/>
             <option type="store" name="repository-type" value="disk"/>
             <option type="store" name="repository-size-threshold" value="104857600"/>
             <option type="store" name="repository-size-limit" value="209715200"/>
             <option type="store" name="repository-disk-file-size-limit" value="1073741824"/>
             <option type="store" name="repository-allow-ack-on-reception" value="1"/>
             <option type="store" name="repository-disk-write-delay" value="1000"/>
             <option type="store" name="receiver-new-registration-rollback" value="0"/>
             <option type="store" name="source-activity-timeout" value="120000"/>
             <option type="store" name="receiver-activity-timeout" value="30000"/>
             <option type="store" name="retransmission-request-forwarding" value="0"/>
          </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
</ume-store>
```

### 4.2.7. RPP Cross Feature Functionality

| UM Feature | Supported | Notes |
|---|---|---|
| **UMP** Proxy Sources | Yes | |
| **UM** Gateway | No | Source-paced Persistence and Receiver-paced Persistence are currently not supported over a **UM** Gateway. |
| **UM** Transports | Yes | |
| Multi-Transport Threads | No | |
| Off-Transport Recovery | Yes | |
| Late Join | Yes | With the new store option, Acknowledge on Reception, sources may not retain sufficient sent messages to provide an effective Late Join capability. |
| HF | Yes | |
| HFX | Yes | |
| Wildcard Receivers | Yes | |

| UM Feature | Supported | Notes |
|---|---|---|
| Message Batching | Yes | |
| Ordered Delivery | Yes | |
| Request/Response | Yes | |
| Multicast Immediate Messaging (MIM) | No | MIM messages are not persisted and have no impact on RPP. |
| Source Side Filtering | Yes | |
| Self-Describing Messaging (SDM) | Yes | |
| Pre-Defined Messaging (PDM) | Yes | |
| **UM** Spectrum | Yes | |
| Monitoring/Statistics | Yes | |
| Acceleration - DBL | Yes | |
| Acceleration - UD | Yes | |
| Implicit/Explicit Acknowledgements | Yes | |
| Registration ID/Session Management | Yes | |
| Fault Tolerance - Round Robin | No | If a RPP source attempts to register to a source repository configured for Round Robin fault tolerance, lbm_src_create() returns an error. |
| Fault Tolerance - Quorum Consensus | Yes | |
| **UM** SNMP Agent | Yes | |
| **Ultra Messaging Manager** | Yes | |
| **Ultra Messaging Cache** | Yes | |
| **Ultra Messaging Desktop Services** | No | |

## 4.3. Queuing Operations

The communication between source and receiver applications and **UMQ** instances follow many of the same patterns as the communication within **UMP** persistence applications. However to reduce the configuration of queue sources and receivers, some Queue-specific activities such as registration or queue resolution operate automatically and become active when the following two conditions occur.

1. The **UM** Configuration file contains the option, umq_queue_name (../Config/ultramessagingqueuingoptions.html#SOURCEUMQQUEUENAME).

2. A `umestored` is running configured for queuing. A queue specification contains not only queue information, but also the application sets receiving queue messages along with the topics being sent to the queue.

This section discusses the following topics.

- *Registration and Assignment*
- *Message Paths*
- *Queue Feedback*
- *UMQ Flight Size*
- *Topics and Queues*
- *Queue Resolution*

## 4.3.1. Registration and Assignment

Source and receiver applications in a queuing operation use the same registration and registration response behaviors as sources and receivers in a persistence operation. (See *UMP Registration*.) The contents of the actual messages for registration and registration response for queues are slightly different, but the message paths are essentially the same.

After a receiver registers with a Queue, the Queue issues an Assignment ID, which identifies the receiver. The Queue assigns this ID to all messages queued for the receiver's topic. The Assignment feature implements the OAOO delivery feature. The Queue assigns messages to a receiver after any one of the following events.

- A message arrives at the Queue. (If no receiver interested in the topic has registered, the Queue holds the message until a receiver interested in the message's topic registers with the Queue.)
- A receiver registers with the Queue. (Obviously, the Queue must possess messages for the receiver's topic.)
- The Queue receives a Consumption Report from a receiver.

### *4.3.1.1. Source Application Registration*

At a high level, the following registration activity occurs before a source application submits messages to a queue topic.

1. umq_queue_name (../Config/ultramessagingqueuingoptions.html#SOURCEUMQQUEUENAME) must be set for queue *Q1*.
2. `umestored` must be started and configured for queue *Q1* and topic *SubjectA*.
3. The source application creates a source object for topic *SubjectA*.
4. The source application's context automatically registers with *Q1* if not already registered.
5. The Queue sends a `Queue Registration Complete` context event to the source application.
6. The source object automatically registers as *SubjectA* with *Q1*.
7. The Queue sends a `Queue Registration Complete` source event to the source application.

#### *4.3.1.2. Receiver Application Registration*

At a high level, the following registration activity occurs before a receiver application can receive queue topic messages.

1. `umestored` must be started and configured for queue *Q1* and topic *SubjectA*.
2. The receiver application creates a receiver object for topic *SubjectA*.
3. The receiver discovers through topic resolution that *Q1* has messages for topic *SubjectA*. (Topic advertisements in a queuing operation contain the queue name.)
4. The receiver application's context automatically registers with *Q1* if not already registered.
5. The Queue sends a `Queue Context Registration Complete` message to the receiver application after the receiver application's context registers.
6. The receiver object automatically registers as *SubjectA* with *Q1* If not using **UMQ** Sessions IDs, the receiver includes its `Assignment ID` with the registration request.
7. The Queue sends a `Queue Receiver Registration Complete` message to the receiver application. If using **UMQ** Sessions IDs, the Queue includes the receiver's `Assignment ID` with the registration complete message.

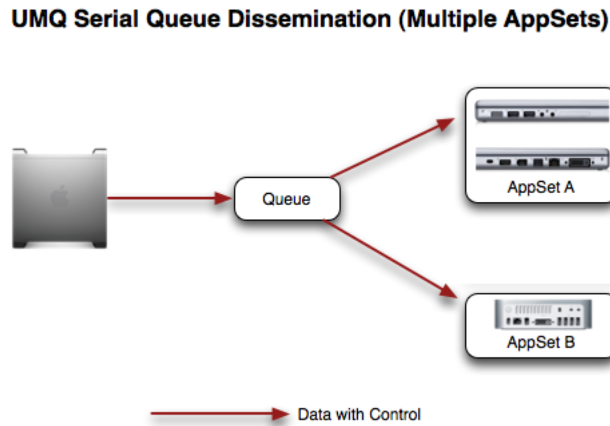
### 4.3.2. Message Paths

Depending on the data dissemination model in use, the messaging paths between sources, receivers, and queue instances may vary quite a bit.

#### *4.3.2.1. Serial Queue Dissemination (SQD)*

The Serial Queue Dissemination (SQD) model (*Serial Queue Dissemination (SQD)*) uses direct serial unicast from the queue to the individual receivers. Receivers only receive the messages they are assigned to process. The term *serial* indicates that the queue sends each message via unicast only to the message's assigned receivers (one in each application set). This dissemination model creates less work for receivers than either PQD or SD, which require receivers to decipher control information to determine the messages they must consume.

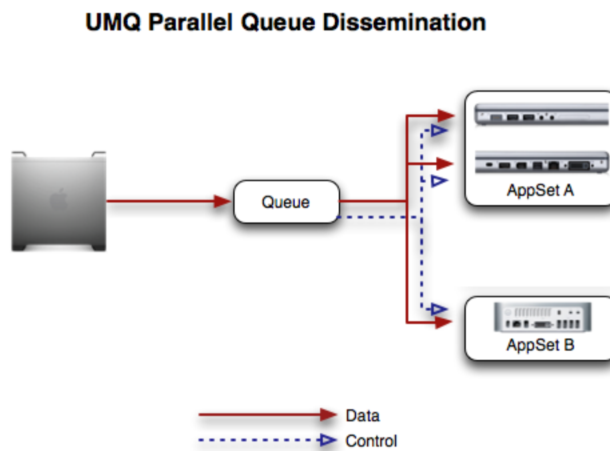
**Figure 12. Serial Queue Dissemination (SQD)**




#### *4.3.2.2. Parallel Queue Dissemination (PQD)*

The Parallel Queue Dissemination (PQD) model (*Parallel Queue Dissemination (PQD)*) uses normal **UM** transport sessions to disseminate messages. In fact, the queue uses individual **UM** topics to send messages to all receivers. In addition, the Queue sends control information (Receiver Control Record - RCR) over a specific topic, configured with the `control-topic-name` Queue option in the Queue's XML configuration file. (See *Queue Element*. Receivers listen to all data and control information and deliver all their assigned messages to the application, ignoring all other messages.

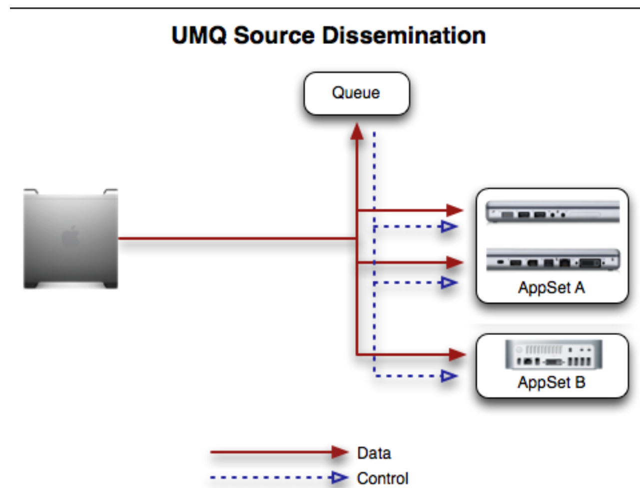
**Figure 13. Parallel Queue Dissemination (PQD)**



#### 4.3.2.3. Source Dissemination (SD)

The *Source Dissemination (SD)* model also uses normal **UM** transport sessions from source application to send message data to the receivers. The queue sends control information (Receiver Control Record - RCR) as in the PQD model that instructs receivers via assignments what to process and what to ignore. However, the queue does not send data messages on topics.

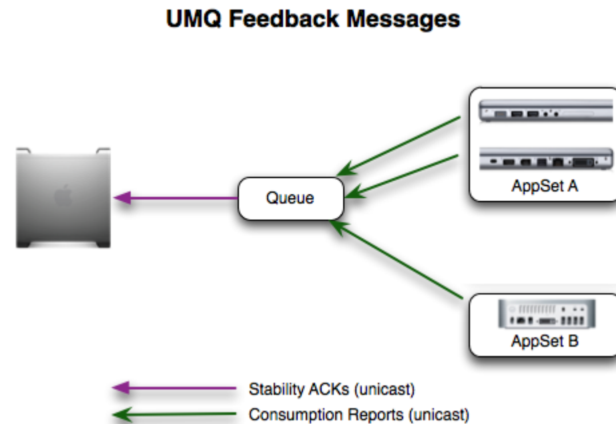
**Figure 14. Source Dissemination (SD)**




### 4.3.3. Queue Feedback

Receiver applications as well as queue instances provide various forms of feedback. Queue instances send Stability Acknowledgements directly back to **UMQ** source applications to indicate successful submission of messages to the queue. Receiver applications signal message consumption by sending Consumption Reports back to the queue instances. See *Queue Feedback*.

**Figure 15. Queue Feedback**




## 4.3.4. UMQ Flight Size

**UMQ** supports a flight size mechanism similar to *UMP Flight Size* that tracks messages in flight from a particular source and responds when a send would exceed the configured flight size (umq_flight_size (../Config/ultramessagingqueuingoptions.html#SOURCEUMQFLIGHTSIZE)). You can configure umq_flight_size_behavior (../Config/ultramessagingqueuingoptions.html#SOURCEUMQFLIGHTSIZEBEHAVIOR) to either:

- block any sends that would exceed the flight size or,
- allow the sends while notifying your application.

**UMQ** considers a sent message in-flight until the source receives the configured number of stability acknowledgements from the queue(s). (No delivery confirmation exists in **UMQ**.) As with **UMP** Quorum/Consensus, intragroup (../Config/ultramessagingqueuingoptions.html#SOURCEUMQRETENTIONINTRAGROUPSTABILITYBEHAVIOR) and intergroup (../Config/ultramessagingqueuingoptions.html#SOURCEUMQRETENTIONINTERGROUPSTABILITYBEHAVIOR) stability settings affect whether **UMQ** considers a messages in flight.

**UMQ** also supports a flight size mechanism for Multicast Immediate Messages (MIM). You configure MIM flight size with the context scope configuration options, (context) umq_flight_size (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQFLIGHTSIZE) and (context) umq_flight_size_behavior (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQFLIGHTSIZEBEHAVIOR).

> **Note:** A source can be configured to publish via **UMP** and **UMQ**. In either of these cases, flight sizes and behaviors can be configured differently with the appropriate configuration options. If a source publishes via both **UMP** and **UMQ** and the flight size behaviors for each are set to block, a send that exceeds **either** flight size will block.

### 4.3.5. Topics and Queues

Sources send messages on topics and receivers listen on topics for messages. Similarly, **UMQ** source applications submit messages to queues with each message being sent on a specific topic. **UMQ** receiver applications listen on topics for messages. This means that Application Sets and the once-and-only-once (OAOO) behavior are on a per topic basis. A potentially helpful analogy is a queue may be the name of a mailbox and a topic may be the subject of an individual letter.

### 4.3.6. Queue Resolution

Unlike **UMP** persistent messaging, queues use a queue resolution mechanism built upon **UM** topic resolution for service location. A **UMQ** source application does not have to know the IP address and TCP port for each queue instance. Instead, all it requires is the name of the queue. This provides some flexibility in deployment for moving queue instances around and requires much less static information to be maintained for configuration of sources and receivers.

### 4.3.7. Queue Browser

**UMQ** supports the JMS Queue Browser specification with the following C API (../API/index.html) and Java API (../JavaAPI/html/index.html) calls.

| Action | C API | Java API |
|---|---|---|
| Retrieve a list of topics and application sets from a running queue daemon. | `lbm_ctx_umq_queue_topic_list()` | `LBMContext.queueTopicList()` |
| Retrieve a list of currently-enqueued message IDs from a running queue daemon. | `lbm_rcv_umq_queue_msg_list()` | `LBMReceiver.queueMessageList()` |
| Retrieve specific messages by message ID from a running queue daemon. | `lbm_rcv_umq_queue_msg_retrieve()` | `LBMReceiver.queueMessageRetrieve()` |

The following **UM** configuration options apply to Queue Browsing.

- `create_queue_browser_context` (../Config/ultramessagingjmsoptions.html#CONNECTIONFACTORYCREATEQUEUEBROWSERCONTEXT)
- `queue_browser_creation_delay` (../Config/ultramessagingjmsoptions.html#CONNECTIONFACTORYQUEUEBROWSERCREATIONDELAY)
- `queue_browser_timeout` (../Config/ultramessagingjmsoptions.html#CONNECTIONFACTORYQUEUEBROWSERTIMEOUT)

#### *4.3.7.1. Queue Browser Authentication*

**UM** queues can authenticate **UM** applications using the Ultra Messaging JMS Queue Browser feature. **UM** applications can also authenticate the queue.

**Note:** Ultra Messaging JMS users do not need to configure authentication for a Queue Browser. **UMQ** uses a internal default user, `jmsuser`, for JMS applications which requires no configuration.

The use of any of the queue browser APIs mentioned in *Queue Browser*, initiates authentication automatically between your application and the queue. You can require authentication between your application and the queue, ensuring that authentication must be successful before queue browsing can occur.

You require authentication by setting the **UM** Configuration option, `umq_require_queue_authentication` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQREQUIREQUEUEAUTHENTICATION), to the default setting of **1** (authentication required). In addition, set the queue (`umestored`) configuration option, `require-client-authentication` to the default value of **1**.

| **If you set ...** | **Then,** |
|---|---|
| `umq_require_queue_authentication` = 1 and `require-client-authentication` = 0 | Authentication fails. (Possible error code, Core-5990-1) Your application does not respond to browsing command responses sent by the queue. |
| `umq_require_queue_authentication` = 0 and `require-client-authentication` = 1 | Authentication fails. (Possible error code, CoreApi-5688-4135) Your application may send queue browsing commands to the queue, but the queue responds with an authentication failure. |
| `umq_require_queue_authentication` = 0 and `require-client-authentication` = 0 | Authentication can either succeed or fail without effect. Queue Browsing occurs. |

#### *4.3.7.2. Setting Queue Browser Authentication Credentials*

Perform the following two tasks to set the Queue Browser authentication credentials.

1. Use the C API `lbm_auth_set_credentials()` or the Java API `LBMAuthUserInfo()` in your application to create users and passwords in your application. A Credential callback provides a way for you to supply alternate credentials in the event of authentication failure.
2. Generate a `password.xml` file that contains the usernames and passwords used by your application. Place `password.xml` in the directory configured in the `umestored` XML configuration file with the option, `lbm-password-file`. See *Daemon Element*. The queue accesses this file during authentication to verify usernames and passwords. You can generate `password.xml` in one of the two following ways.
   - Use the `lbm_authstorage_*()` API calls in an auxiliary application to create `password.xml`. This application can import any existing user credentials (i.e. from LDAP). (No Java equivalent exists for these functions.)
   - Use the **UMQ** utility, `/bin/lbmpwdgen` to generate `password.xml`. Usage information appears within the file. (This utility uses `lbm_authstorage_*()` API calls.)

A sample `password.xml` appears below. Notice that you can also create and assign user roles. In the sample you may also notice that `/bin/lbmpwdgen` creates an anonymous user (`<user name="">`). **UMQ** requires this user when authentication has not been enabled. You should not delete or edit this user.

```
<?xml version="1.0"?>
<um-configuration version="1.0">
  <users>
    <user name="userSmith">
      <verifier>69A4aAE70US3NiuZr/TvAbSWztu5na5TbFo8bdHxU5.ILdMu8rLd5ragE3p4Qcuz/nXxAj6kGnwIF2JrKdCf
        <salt>3BmRWUznvzy0n2</salt>
        <roles>
          <role name="admin"></role>
          <role name="normal"></role>
        </roles>
    </user>
    <user name="">
      <verifier/>
      <salt/>
      <roles>
        <role name="admin"/>
      </roles>
    </user>
  </users>
  <roles>
    <role name="admin">
      <action>MSG_LIST</action>
      <action>MSG_RETRIEVE</action>
      <action>TOPIC_LIST</action>
     </role>
    <role name="normal">
      <action>TOPIC_LIST</action>
    </role>
  </roles>
</um-configuration>
```

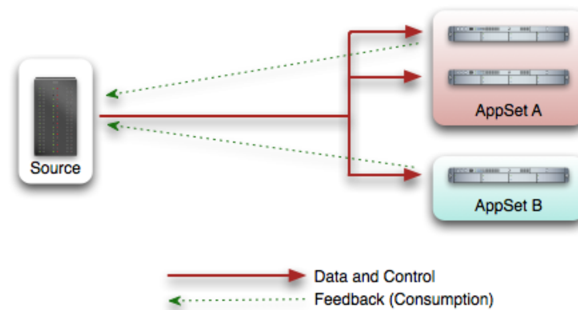
## 4.4. Ultra Load Balancing Operations

ULB, while similar in some ways to **UMQ**, differs in some other ways. This section provides some details on how ULB works, and some comparisons with **UMQ**.

### 4.4.1. Parallel Source Dissemination (PSD)

ULB supports the Parallel Source Dissemination (PSD) dissemination model. In the figure below, note that the ULB PSD model operates in a very similar way to the **UMQ** *Parallel Queue Dissemination (PQD)* model.

**Figure 16. ULB Parallel Source Dissemination (PSD)**




The key difference between ULB's PSD model and **UMQ**'s is that with ULB, the source sends both the data and the assignment/control information on the same transport session. ULB has no need for a control channel, as with **UMQ**.

ULB receivers, as with **UMQ**, send unicast Consumption Reports back to the source.

### 4.4.2. Source Fairness

ULB differs from **UMQ** somewhat in source "fairness". While **UMQ** decouples sources from receivers with a queue -- and therefore avoids receiver overload -- ULB does not use a queue or other middleware, so sources could conceivably combine to overload a receiver, because of the real-time nature of low-latency load balancing.

Therefore, ULB governs fairness between sources by using a) receiver feedback information and b) receiver portion size, in concert with c) source configuration option umq_ulb_application_set_load_factor_behavior (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBAPPLICATIONSETLOADFACTORBEHAVIOR). For more, see *Queue Feedback* and *Receiver Portion Size*.

### 4.4.3. Ultra Load Balancing Flight Size

ULB supports a flight size mechanism similar to *UMP Flight Size* and *UMQ Flight Size* that tracks messages in flight from a particular source and responds when a send would exceed the configured flight size (umq_ulb_flight_size (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBFLIGHTSIZE)). You can configure umq_ulb_flight_size_behavior (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBFLIGHTSIZEBEHAVIOR) to either:

- block any sends that would exceed the flight size or,
- allow the sends while notifying your application.

ULB considers a message in-flight until the source receives the MSG_COMPLETE notification from all application sets. See also umq_ulb_events (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBEVENTS).

### 4.4.4. Indexed Ultra Load Balancing (ULB)

Indexed Ultra Load Balancing (ULB) is similar to *Indexed Queuing*. The source and receiver APIs are exactly the same with only minor differences (e.g., receiver event types remain the same as for indexed queueing, but have a flag set indicating that the source of the event was a ULB source). There are two important differences to note:

- Assignment of indices is done at the source, on a per-source basis, with no global coordination of index assignments, so multiple sources sending on the same index may happen to assign the index to different receivers.
- The advanced `allow/deny` configuration rules and configuration option inheritance available via the `umestored's` *Index Rules Element* for indexed queueing are not available for indexed ULB.

## 4.5. UMP and UMQ Events

The **Ultra Messaging** API provides a number of events, callbacks, messages, functions, and settings. The API reference (C API (../API/index.html), Java API (../JavaAPI/html/index.html) or .NET API (../DotNetAPI/doc/Index.html)) can be used to see the true extent of the API. In order to design successful applications, though, a high level understanding of the events and callbacks is essential.

- **Events** - Source events occur on a per source basis.
- **Callbacks** - Source and receiver callbacks called directly from **UMP** and **UMQ** internal operation and usually demands a return value be filled in and/or are informational in nature. Typically, applications do very little processing in callbacks.
- **Messages** - Messages to receivers can simply contain **UMP** or **UMQ** information or have impact on operation.

Some specific languages, such as C, Java, or C# may have specific nuances for the various events and callbacks. But, by and large, an application should plan on having access to the items listed in the following sections. For details for a particular language, consult the **Ultra Messaging** API documentation (C API (../API/index.html), Java API (../JavaAPI/html/index.html) or .NET API (../DotNetAPI/doc/Index.html)).

### 4.5.1. Source Events

The following events and callbacks are available for source applications.

| Event Name | Type | Description |
| --- | --- | --- |
| `Store Registration Success` | Source Event | Delivered once a source has successfully registered with a single store. Event contains flags to show if the source is "old" (i.e. a re-registration) as well as the sequence number that the source should use as its initial sequence number when sending, and the store information |

| Event Name | Type | Description |
| --- | --- | --- |
| `Store Registration Complete` | Source Event | Delivered once a source has completed registration with the required store(s). This indicates the source may send as it desires. Event contains the consensus sequence number. |
| `Store Registration Error` | Source Event | Delivered once a source has received an error from the store indicating the requested registration was not granted. Event contains an error message to indicate what happened. |
| `Queue Registration Complete` | Source Event | Delivered once a source has completed registration with the Queue. This indicates the source may send as it desires. Event contains the Registration ID or Session ID. |
| `Queue Registration Error` | Source Event | Delivered once a source has received an error from the Queue indicating the requested registration was not granted. Event contains an error message to indicate what happened. |
| `Store Message Stable` | Source Event | Delivered once a message is stable at a single store. Event contains the message sequence number and indicates if the the message meets Intergroup and/or Intragroup stability requirements. Also includes the store information. |
| `Queue Message Stability ACK` | Source Event | Stability acknowledgement sent to the source application by the Queue when a message sent by the source is stable. If the Queue is configured to persist data to disk, the message is stable once it has been written to the file. Otherwise, the Queue acknowledges the message upon receipt. |
| `Queue Message ID Information` | Source Event | The Queue assigns a unique ID to every message in order to assign the message to a receiver. This assignment enforces OAOO delivery. The Queue includes assignment information in the control information sent to receivers in a Parallel Queue Dissemination (PQD) or Source Dissemination (SD) model. |

| Event Name | Type | Description |
|---|---|---|
| `Delivery Confirmation` | Source Event | Delivered once a message has been confirmed as delivered and processed by a receiving application. Event contains the message sequence number as well as indications whether the message has met the unique confirmations requirement. Also contains the receiver's Registration ID or Session ID. |
| `Store Unresponsive` | Source Event | Delivered once a store is seen to be unresponsive due to failure or network disconnect. Event contains a message with more details suitable for logging. Sources using the unresponsive store as their only store (not in Round-Robin or Quorum/Consensus) will be prevented from sending until the store recovers. |
| `Store Message Reclaimed` | Source Event | Delivered once a message has passed through retention and is about to be released from memory or disk. Event contains the message sequence number. (Reclaim refers to storage space reclamation.) |
| `Store Forced Reclaim` | Callback | Indicates a message is being forcibly released because the memory size limit ( retransmit_retention_size_limit (../Con- fig/latejoinoptions.html#SOURCERETRANSMITRETE has been exceeded. Event contains the message sequence number. |
| `Flight Size Notification` | Callback | Indicates that the number of in-flight messages for a source has exceeded or fallen below the configured flight size limit for a source. The event indicates if the flight size has been exceeded (OVER) by a new message send or that a message recently stabilized has reduced the number of in flight messages to less than the flight size limit (UNDER). Notification also indicates if the event is for **UMP**, **UMQ** or ULB. |

| Event Name | Type | Description |
| --- | --- | --- |
| `RPP Source Registration Success` | Source Event | Delivered once a source has successfully registered with a single store as a RPP source. The event contains either the RegID or Session ID, the sequence number of the last message stored for the source and store information. |
| `RPP Source Registration Failure` | Source Event | Delivered once a source has received an error from the store indicating the requested registration was not granted. Event contains an error message to indicate what happened. |
| `RPP Source Deregistration Success` | Source Event | Delivered once a source successfully deregisters from an individual store. The event contains either the RegID or Session ID, the sequence number of the last message stored for the source and store information. |
| `RPP Source Deregistration Complete` | Source Event | Delivered once UMP receives a successful deregistration event from all stores. |

### 4.5.2. Receiver Events

The following callbacks and messages are available for receiver applications

| Event Name | Type | Description |
| --- | --- | --- |
| `Store Registration Success` | Message | Delivered once a receiver has successfully registered with a single store. Message contains flags to show if the receiver is "old" (i.e. Not a new registration) as well as the sequence number that the receiver should use as its low sequence number, and the store information. In addition, the event contains the source's Registration ID or Session ID and the receiver's Registration ID or Session ID. |
| `Store Registration Complete` | Message | Delivered once a receiver has completed registration with the store(s) required. This indicates the receiver may now receive data. Message contains the consensus sequence number. |

| Event Name | Type | Description |
| --- | --- | --- |
| `RPP Receiver Registration Success` | Message | Delivered once a receiver has successfully registered with a single store as a RPP receiver. Message contains either the RegID or Session ID, the sequence number of the last message stored for the source and store information. |
| `RPP Receiver Registration Failure` | Message | Delivered once a receiver has received an error from the store indicating the requested registration was not granted. Event contains an error message to indicate what happened. |
| `RPP Receiver Deregistration Success` | Message | Delivered once a receiver successfully deregisters from an individual store. The message contains either the RegID or Session ID for the receiver and the source, the sequence number of the last message stored for the source and store information. |
| `RPP Receiver Deregistration Complete` | Message | Delivered once UMP receives a successful deregistration event from all stores. |
| `Queue Registration Complete` | Message | Delivered once a receiver has completed registration with the Queue. Message contains assignment information. |
| `Store Registration Error` | Message | Delivered once a receiver has received an error from the store indicating the requested registration was not granted. Message contains an error message to indicate what happened. |
| `Queue Registration Error` | Message | Delivered once a receiver has received an error from the Queue indicating the requested registration was not granted. Message contains an error message to indicate what happened. |
| `Store Registration Change` | Message | Delivered once a change in store information is received from the source. The extent of the change is included in a message suitable for logging. |

| Event Name | Type | Description |
|---|---|---|
| `Queue Deregistration Complete` | Message | Event delivered to a **UMQ** receiving application by the Queue when a receiver deregisters from a Queue by calling `lbm_rcv_umq_deregister()`. All messages must be delivered to the application before the Queue sends the deregistration event. Deregistered receivers cannot be assigned any new messages. |
| `Store Retransmission` | Message | Retransmissions from recovery come in as normal messages with a flag indicating their status as a retransmission. |
| `Store Registration Function` | Callback | Called once a receiver receives store information from a source and **UMP** desires to know the RegID to use for the receiver. Callback passes the source RegID, the store information, and the source transport name. The return value is the RegID that **UMP** should request to use from the store. |
| `Store Recovery Sequence Number Function` | Callback | Called once registration is about to complete and the low sequence number must be determined. Callback passes the highest sequence number seen from the source and the consensus sequence number from the stores or sequence number from the store if using round-robin. |
| `Queue Index Assignment Eligibility Start Complete` | Message | Event delivered to a **UMQ** receiving application by the Queue when a receiver becomes eligible for new index assignments from a Queue as a result of calling the `lbm_rcv_umq_index_start_assignment()` function. |

| Event Name | Type | Description |
|---|---|---|
| `Queue Index Assignment Eligibility Stop Complete` | Message | Event delivered to a **UMQ** receiving application by the Queue when a receiver becomes ineligible for new index assignments from a Queue as a result of calling the `lbm_rcv_umq_index_stop_assignment()` function. After this event is delivered, a receiver no longer receives new index assignments from the Queue. Existing index assignments remain in place. |
| `Queue Index Assigned` | Message | Event delivered to a **UMQ** receiving application by the Queue when the Queue assigns a receiver a new index. After this event is delivered, the receiver begins receiving messages on the new index. |
| `Queue Index Released` | Message | Event delivered to a **UMQ** receiving application by the Queue when the Queue removes one of the receiver's existing index assignments. This usually occurs after the receiver calls `lbm_rcv_umq_index_release()`. As a result, the receiver no longer receives any messages for the removed index. |

### 4.5.3. Context Events

The following events are available for the context of source and receiver applications.

| Event Name | Type | Description |
|---|---|---|
| `Queue Registration Complete` | Context Event | Delivered once a source or receiver application's context has completed registration with the Queue. A context only needs to register once with a Queue. Event contains the Registration ID or Session ID. |
| `Queue Registration Error` | Context Event | Delivered once a source or receiver application has received an error from the Queue indicating the requested registration was not granted. Event contains an error message indicating what happened. |

| Event Name | Type | Description |
|---|---|---|
| `Instance List Notification` | Context Event | Delivered once a queue instance has changed. Event holds information string. |
| `Flight Size Notification` | Context Event | Indicates that the number of in-flight Multicast Immediate Messages has exceeded or fallen below the configured flight size limit. The event indicates if the flight size has been exceeded (OVER) by a new message send or that a message recently stabilized has reduced the number of in flight messages to less than the flight size limit (UNDER). |

# 5. Enabling Persistence

In this section, we explain how to build a persistence messaging application by starting with a minimum source and receiver and then adding **UMP** features incrementally. With the help of example source, this section explains the following operations.

- *Adding the UMP Store to a Source*
- *Adding Fault Recovery with Registration IDs*
- *Enabling Persistence Between the Source and Store*
- *Enabling Persistence in the Source*
- *Enabling Persistence in the Receiver*

Prerequisite: You should understand basic **Ultra Messaging** concepts such as Sources and Receivers and the basic methods for configuring them.

The following table lists all source files used in this section. You can also find links to them in the appropriate task. The files can also be found in the `/doc/UME` directory.

| Object | Filename |
|---|---|
| Source Application | ume-example-src.c |
| Receiver Application | ume-example-rcv.c |
| Source Application 2 | ume-example-src-2.c |
| Receiver Application 2 | ume-example-rcv-2.c |
| Source Application 3 | ume-example-src-3.c |
| Receiver Application 3 | ume-example-rcv-3.c |
| **UMP** Store Configuration File | ume-example-config.xml |

## 5.1. Starting Configuration

We begin with the minimal source and receiver used by the QuickStart Guide (../QuickStart/index.html). To more easily demonstrate the **UMP** features we are interested in, we have modified the QuickStart source and receiver in the following ways.

- Modified the source to send 20 messages with a one second pause between each message
- Modified the receiver to anticipate 20 messages instead of just one
- Assigned the topic, UME Queue Example, to both the source and receiver
- Modified the receiver to not exit on unexpected receiver events

The last change allows us to better demonstrate basic operation and evolve our receiver slowly without having to anticipate all the options that **UMP** provides up front.

Example files for our exercise are:

| Object | File |
|---|---|
| Source Application | ume-example-src.c |
| Receiver Application | ume-example-rcv.c |

> **Note:** Be sure to build `ume-example-rcv.c` and `ume-example-src.c`. Instructions for building them are at the beginning of the source files.

## 5.2. Adding the UMP Store to a Source

The fundamental component of a **UMP** persistence solution is the persistent store. To use a store, a source needs to be configured to use one by setting `ume_store` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTORE) for the source. We can do that with the following piece of code.

```
err = lbm_src_topic_attr_str_setopt(&attr, "ume_store", "127.0.0.1:14567");
```

This sets the **UMP** persistent store for the source to the store running at 127.0.0.1 on port 14567.

> **Note:** If you desire to run a store on a different machine than where the source and receiver are run, then you should replace 127.0.0.1 with the IP address (not hostname) of the machine running the **UMP** persistent store.

Example files for our exercise are:

| Object | Filename |
|---|---|
| Source Application | ume-example-src.c |
| Receiver Application | ume-example-rcv.c |
| **UMP** Store Configuration File | ume-example-config.xml |

After adding the ume-store specification to the source, perform the following steps.

1. Create the cache and state directories. `$ mkdir umestored-cache ; mkdir umestored-state`
2. Start up the store. `$ umestored ume-example-config.xml`
3. Start the Receiver. `$ ume-example-rcv`
4. Start the Source. `$ ume-example-src`

You should see a message on the source that says:

```
INFO: Source "UME Example" Late Join not set, but UME store specified. Setting Late Join.
```

This is an informational message from **UMP** and merely means Late Join was not set and that **UMP** is going to set it.

Notice that the receiver was not configured with any store information. That is because setting it on the source is all that is needed. The receiver learns **UMP** store settings from the source through the normal **UM** topic resolution process. Receivers don't need to do anything special to leverage the usage of a store by a source.

## 5.3. Adding Fault Recovery with Registration IDs

If the source or receiver crashes, how does the source and receiver tell the store that they have restarted and wish to resume where they left off? We need to add in some sort of identifiers to the source and receiver so that the store knows which sources and receivers they are.

In **UMP** , these identifiers are called Registration IDs or RegIDs. **UMP** allows the application to control the use of RegIDs as it wishes. This allows applications to migrate sources and receivers not just between systems, but between locations with true, unprecedented freedom. However, **UMP** requires an application to be careful of how it uses RegIDs. Specifically, an application must not use the same RegID for multiple sources and/or receivers at the same time.

Now let's look at how we can use RegIDs to provide complete fault recovery of sources and receivers. We'll first handle RegIDs in the simplest manner by using static IDs for our source and receiver. For the source, the RegID of 1000 can be added to the existing store specification by changing the string to

```
127.0.0.1:14567:1000
```

This yields the source code in ume-example-src-2.c

For the receiver, we accomplish this in two steps.

1. Set a callback function to be called when we desire to set the RegID to 1100. This is done by declaring the callback function, `app_rcv_regid_callback`, which will return the RegID value 1100 to **UMP** .
2. Inform the **UMP** configuration for the receiver to use this callback function. That is accomplished by setting the `ume_registration_extended_function` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEREGISTRATIONEXTENDEDFUNCTION) similar to example code below.

```
lbm_ume_rcv_regid_ex_func_t id;      /* structure to hold registration function information */
id.func = app_rcv_regid_callback;    /* the callback function to call */
id.clientd = NULL;                   /* the value to pass in the clientd to the function */
err = lbm_rcv_topic_attr_setopt(&attr, "ume_registration_extended_function", &id, sizeof(id));
```

Once this is done, the receiver has the ability to control what RegID it will use. This yields the source code in ume-example-rcv-2.c.

With these in place, you can experiment with killing the receiver and bringing it back (as long as you bring it back before the source is finished), as well as killing the source and bringing it back.

The restriction to this initial approach to RegIDs is that the RegIDs 1000 and 1100 may not be used by any other objects at the same time. If you run additional sources or receivers, they must be assigned new RegIDs, not 1000 or 1100. Let's now take a more sophisticated approach to RegIDs that will allow much more flexibility

## 5.4. Enabling Persistence Between the Source and Store

Let's refine our source to include some desired behavior following a crash. Upon restart, we want our source to resume with the first unsent message. For example, if the source sent 10 messages and crashed, we want our source to resume with the 11th message and continue until it has sent the 20th message.

Accomplishing this graceful resumption requires us to ensure that our source is the only source that uses the RegID assigned to it. The same RegID should be used as long as the source has not sent the 20th message regardless of any crashes that may occur. We can do this with the following changes to the store:

1. Configure the store to assign a RegID when the source starts.
2. Configure the store to save the RegID to disk so that it can be used after a crash.

In addition to these two changes to the store's configuration, the following two sections explain the changes needed for the source and receiver, which become fairly easy due to the events that **UMP** delivers to the application during **UMP** operation.

> **Note:** While the following sections are instructive about how **UMP** uses RegIDs to provide persistence, RegIDs can also be managed easily with the use of Session IDs. See *Managing RegIDs with Session IDs*.

## 5.5. Enabling Persistence in the Source

With the above mentioned behaviors in mind, let's turn to looking at how they may be implemented with **UMP** , starting with the source. We can summarize the changes we need by the following list.

1. At source startup, use any saved RegID information found in the file by setting information in the `ume_store` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTORE) configuration variable.
2. After the store registration is successful, if a new RegID was assigned to the source, save the RegID to the file.
3. Set the message number to begin sending. Refer to the explanation below.
4. Send until message number 20 has been sent.
5. After message 20 has been sent, delete the saved RegID file.

For Step 3, if the source has just been initialized, the application starts with message number 1. If the source has been restarted after a crash, the application looks to **UMP** to establish the beginning message number because **UMP** will use the next sequence number. For this simple example, we can make the assumption that each message is one sequence number for **UMP** and that **UMP** starts with sequence number 0. Thus the application can set the message number it begins resending with the value of the **UMP** sequence number + 1.

> **Note:** Using sequence numbers to set the message number is a good practice if you send messages smaller than 8K.

These changes yield the source code in ume-example-src-3.c.

## 5.6. Enabling Persistence in the Receiver

Let's also refine the receiver to resume where it left off after a crash. Just as with the source, the receiver can have the store assign it a RegID if the receiver is just beginning. Once the receiver receives the 20th message from the source, it can get rid of the RegID and exit. Because the receiver can receive some messages, crash, and come back, we should only need to look at a message and check if it is the 20th message based on the message contents or sequence number. **UMP** provides all the events to the application that we need to create these behaviors in the receiver.

The receiver changes are summarized below.

1. At receiver startup, use any saved RegID information found in the file for callback information when needed.
2. When RegID callback is called: Check to see if the source RegID matches the saved source RegID. If it does, return the saved receiver RegID. RegID matches the saved source RegID if so, return the saved receiver RegID.
3. After store registration is successful: If not using a previously saved RegID, then save the RegID assigned by the store to the source to a file, as well as the store information and the source RegID.
4. After the last message is received (message number 20 or **UMP** sequence number 19), end the application and delete the saved RegID file.

RegIDs in **UMP** can be considered to be per source and per topic. Thus the receiver does not want to use the wrong RegID for a different source on the same topic. To avoid this, we save the source RegID and even store information so that the `app_rcv_regid_callback` can make sure to use the correct RegID for the given source RegID. These changes yield the source code in ume-example-rcvc-3.c (ume-example-rcv-3.c)

The above sources and receivers are simplified for illustration purposes and do have some limitations. The receiver will only keep the information for one source at a time saved to the file. This is fine for illustration purposes, but would be lacking in completeness for production applications unless it was assured that a single source for any topic would be in use. To extend the receiver to include several sources is simply a matter of saving each to the file, reading them in at startup, and being able to search for the correct one for each callback invoked.

# 6. Demonstrating Persistence

This section demonstrates the following events using the `ume-example` applications described in *Enabling Persistence*.

- *Running UMP Example Applications*
- *Single Receiver Fails and Recovers*
- *Single Source Fails and Recovers*
- *Single Store Fails*

> **Note:** While these four sections demonstrate how **UMP** uses RegIDs to provide persistence, RegIDs can also be managed easily with the use of Session IDs. See *Managing RegIDs with Session IDs*.

The following table lists all source files used in this section. The files can also be found in the `/doc/UME` directory.

| Object | Filename |
|---|---|
| Source Application 3 | ume-example-src-3.c |
| Receiver Application 3 | ume-example-rcv-3.c |
| **UMP** Store Configuration File | ume-example-config.xml |

Perform the following tasks first.

1. Build `ume-example-rcv-3.c` (ume-example-src-3.c) and `ume-example-src-3.c` (ume-example-rcv-3.c). Instructions for building them are at the beginning of the source files.
2. Create default directories, `umestored-cache` and `umestored-state` in the `/doc/UME` directory where the other ume-example files are located. Our sample XML store configuration file, `ume-example-config.xml`, doesn't specify directories for the store's cache and state files, so those will be placed in the default directories.
3. Start the store. `$ umestored ume-example-config.xml`

You should see no output if the store started successfully. However, you should find a new log file, `ume-example-stored.log`, in the directory you ran the store in. The first couple lines should look similar to below.

```
Fri Feb 01 07:34:28 2009 [INFO]: Latency Busters Persistent Store version 2.0
Fri Feb 01 07:34:28 2009 [INFO]: LBM 3.3 [UME-2.0] Build: Jan 31 2009, 02:10:43
( DEBUG license LBT-RM LBT-RU ) WC[PCRE 6.7 04-Jul-2006, appcb]
```

You'll also be able to view the store's web monitor. Open a web browser and go to:

```
http://127.0.0.1:15304/
```

You should see the store's web monitor page, which is a diagnostic and monitoring tool for the **UMP** store. See *Ultra Messaging Web Monitor*.

## 6.1. Running UMP Example Applications

With the store running, let's try our example source and receiver applications.

1. Start the Receiver. `$ ume-example-rcv-3.exe`
2. Start the Source. `$ ume-example-src-3.exe`

You should see output for the source similar to the following:

```
saving RegID info to "UME-example-src-RegID" - 127.0.0.1:14567:2795623327
```

You should see output for the receiver similar to the following:

```
UME Store 0: 127.0.0.1:14567 [TCP:169.254.97.160:14371][2795623327] Requesting RegID: 0
saving RegID info to "UME-example-rcv-RegID" - 127.0.0.1:14567:2795623327:2795623328
Received 15 bytes on topic UME Example (sequence number 0) 'UME Message 01'
Received 15 bytes on topic UME Example (sequence number 1) 'UME Message 02'
Received 15 bytes on topic UME Example (sequence number 2) 'UME Message 03'
Received 15 bytes on topic UME Example (sequence number 3) 'UME Message 04'
...
```

The example source sends 20 messages. After the 20th messages, both the source and receiver exit and print the message `removing saved RegID file...` So what just happened? Let's walk through the output line by line.

**Source**

```
saving RegID info to "UME-example-src-RegID" - 127.0.0.1:14567:2795623327
```

The source successfully registered with the store using its pre-configured store address and port of 127.0.0.1:14567. It didn't ask for a specific RegID from the store, so the store automatically assigned one to it. In this case, the store assigned the ID, 2795623327. Your source's ID will likely be different because stores assign random RegIDs.

If you run the test again, you'll notice the source application has written a file called `UME-example-src-RegID` that contains the same information the source printed on startup, namely the IP address and port of the store it registered with, along with its RegID assigned by the store.

**Receiver**

```
UME Store 0: 127.0.0.1:14567 [TCP:169.254.97.160:14371][2795623327] Requesting RegID: 0
saving RegID info to "UME-example-rcv-RegID" - 127.0.0.1:14567:2795623327:2795623328
```

The receiver has been informed of how to connect to the store by the source, and it also successfully registered with the store. The store's IP address and port are shown, followed by the source's unique identifier string (in this case, it's a TCP source on port 14371), and the source's RegID. The receiver then requests RegID 0 from the store, which is a special value that means *pick an ID for me* (Although not displayed, the source requested ID 0 when it started up as well).

In parallel with the source application, the receiver application writes its RegID with this store to the file, `UME-example-rcv-RegID`.

After sending 20 messages under normal, stable conditions, the source and receiver applications exit and remove their RegID files.

## 6.2. Single Receiver Fails and Recovers

Perform the following procedure with the store running to see what happens when a receiver fails and recovers.

1. Start the Receiver. `$ ume-example-rcv-3.exe`
2. Start the source. `$ ume-example-src-3.exe` Let it run for a few seconds so the receiver gets a few messages.

```
UME Store 0: 127.0.0.1:14567 [TCP:169.254.97.160:14371][3735579353] Requesting RegID: 0
saving RegID info to "UME-example-rcv-RegID" - 127.0.0.1:14567:3735579353:3735579354
Received 15 bytes on topic UME Example (sequence number 0) 'UME Message 01'
Received 15 bytes on topic UME Example (sequence number 1) 'UME Message 02'
Received 15 bytes on topic UME Example (sequence number 2) 'UME Message 03'
```

3. Stop the receiver (**Ctrl/C**) and leave the source running. Wait a few more seconds so that the source sends some messages while the receiver was down.
4. Restart the Receiver and let it run to completion. `$ ume-example-rcv-3.exe`

```
read in saved RegID info from "UME-example-rcv-RegID" - 127.0.0.1:14567 RegIDs
source 3735579353, receiver 3735579354
UME Store 0: 127.0.0.1:14567 [TCP:169.254.97.160:14371][3735579353]
Requesting RegID: 3735579354
Received 15 bytes on topic UME Example (sequence number 3) 'UME Message 04'
Received 15 bytes on topic UME Example (sequence number 4) 'UME Message 05'
Received 15 bytes on topic UME Example (sequence number 5) 'UME Message 06'
Received 15 bytes on topic UME Example (sequence number 6) 'UME Message 07'
Received 15 bytes on topic UME Example (sequence number 7) 'UME Message 08'
Received 15 bytes on topic UME Example (sequence number 8) 'UME Message 09'
Received 15 bytes on topic UME Example (sequence number 9) 'UME Message 10'
Received 15 bytes on topic UME Example (sequence number 10) 'UME Message 11'
```

Notice that the receiver picked up the message stream right where it had left off - after message 3. The first few messages (which the source had sent while the receiver was down) appear to come in much faster than the source's normal rate of one per second. That's because they are being served to the receiver from the store. The remaining messages continue to come in at the normal one-per-second rate because they're being received from the source's live message stream. This is *durable subscription* at work.

## 6.3. Single Source Fails and Recovers

Perform the following procedure with the store running to see what happens when a source fails and recovers.

1. Start the Receiver. `$ ume-example-rcv-3.exe`
2. Start the source. `$ ume-example-src-3.exe` Let it run for a few seconds so the receiver gets a few messages.
3. Stop the Source (**Ctrl/C**).
4. Restart the Source and let it run to completion. `$ ume-example-rcv-3.exe`

**Source**

You should see output similar to the following on the second run of the source.

```
read in saved RegID info from "UME-example-src-RegID" - 127.0.0.1:14567:2118965523
will start with message number 5
removing saved RegID file "UME-example-src-RegID"
```

**Receiver**

The receiver's output looks like the following.

```
UME Store 0: 127.0.0.1:14567 [TCP:169.254.97.160:14371][2118965523] Requesting RegID: 0
saving RegID info to "UME-example-rcv-RegID" - 127.0.0.1:14567:2118965523:2118965524
Received 15 bytes on topic UME Example (sequence number 0) 'UME Message 01'
Received 15 bytes on topic UME Example (sequence number 1) 'UME Message 02'
Received 15 bytes on topic UME Example (sequence number 2) 'UME Message 03'
Received 15 bytes on topic UME Example (sequence number 3) 'UME Message 04'
UME Store 0: 127.0.0.1:14567 [TCP:169.254.97.160:14371][2118965523] Requesting RegID: 2118965524
saving RegID info to "UME-example-rcv-RegID" - 127.0.0.1:14567:2118965523:2118965524
Received 15 bytes on topic UME Example (sequence number 4) 'UME Message 05'
Received 15 bytes on topic UME Example (sequence number 5) 'UME Message 06'
Received 15 bytes on topic UME Example (sequence number 6) 'UME Message 07'
Received 15 bytes on topic UME Example (sequence number 7) 'UME Message 08'
...
```

When the source was restarted, it read in its previously saved RegID and requested the same ID when registering with the store. The store informed the source that it had left off at sequence number 3 (UME Message 04), and the next sequence number it should send is 4 (UME Message 05). Bringing the source back up also caused the receiver to re-register with the store. Receivers can *only* find out about stores from sources they are listening to. Once the receiver re-registered with the store, it continued receiving messages from the source where it had left off.

## 6.4. Single Store Fails

Perform the following procedure with the store running to see what happens when the store itself fails.

1. Start the Receiver. `$ ume-example-rcv-3.exe`
2. Start the source. `$ ume-example-src-3.exe` Let it run for a few seconds so the receiver gets a few messages.
3. Stop the Store (**Ctrl/C**).

Notice that with this simple example program, the source simply prints the following and exits.

```
saving RegID info to "UME-example-src-RegID" - 127.0.0.1:14567:4095035673
Store unresponsive: store 0 [127.0.0.1:14567] unresponsive
Store unresponsive: store 0 [127.0.0.1:14567] unresponsive - no registration response.
line 318: not currently registered with enough UMP stores
```

When a source application tries to send a message without being registered with a store, the send call returns an error. Messages sent while not registered with a store cannot be persisted. See *UMP Stores* for information about using multiple stores.

Your source application(s) should assume an unresponsive store is a temporary problem and wait before sending the message again. See `umesrc.c` (../example/umesrc.c), `umesrc.java` (../java_example/umesrc.java), or `umesrc.cs` (../dotnet_example/umesrc.cs) for examples of this behavior.

# 7. Designing Persistence Applications

This section discusses considerations and methods for utilizing **UMP** persistence in your applications.

- *Pieces of a Persistence Solution*
- *Fault Recovery*

## 7.1. Pieces of a Persistence Solution

In **UMP** , a persistent system is composed of **sources**, **receivers**, and **stores** managed by one or more applications. Sources and receivers are the endpoints of communication and the store(s) provide fault recovery and persistence of state information. Your application can leverage **UMP** 's flexible methods of persistence to add an unprecedented level of fault tolerance. With this flexibility your applications assume new responsibilities not normally required in other persistent messaging systems. This section identifies the important considerations for your messaging applications when implementing the following **UMP** features.

- *Registration Identifiers*
- *UMP Sources*
- *UMP Receivers*
- *UMP Stores*

### 7.1.1. Registration Identifiers

As mentioned in *Registration Identifier* and *Adding Fault Recovery with Registration IDs*, stores use RegIDs to identify sources and receivers. **UMP** offers three main methods for managing RegIDs.

- Your applications assign static RegIDs and ensure that the same RegID is not assigned to multiple sources and/or receivers. See *Use Static RegIDs*.
- You can allow **UMP** stores to assign RegIDs and then save the assigned RegIDs. See *Save Assigned RegIDs*
- Use Session IDs to enable the **UMP** store to both assign and manage RegIDs. See *Managing RegIDs with Session IDs*

Your applications can manage RegIDs for the lifetime of a source or receiver as long as multiple applications do not reuse RegIDs simultaneously on the same store. RegIDs only need to be unique on the same store and may be reused between stores as desired. You can use a static mapping of RegIDs to applications or use some simple service to assign them.

#### *7.1.1.1. Use Static RegIDs*

The simplest method uses static RegIDs for individual applications. This method works best if:

- Applications use separate stores
- Multiple instances of an application also use separate stores

In the latter case, the same static source RegID can be used in every instance of the application because receivers will identify every Store/Source RegID tuple as unique.

The following source code examples assign a static RegID to a source by adding the RegID, `1000`, to the `ume_store` attribute. (See also ume-example-src-2.c.)

**C API**

```
lbm_src_topic_attr_t * sattr;

if (lbm_src_topic_attr_create(&sattr) == LBM_FAILURE) {
        fprintf(stderr, "lbm_src_topic_attr_create: %s\n", lbm_errmsg());
        exit(1);
}
if (lbm_src_topic_attr_str_setopt(sattr, "ume_store", "127.0.0.1:14567:1000")
== LBM_FAILURE) {
        fprintf(stderr, "lbm_src_topic_attr_str_setopt: %s\n", lbm_errmsg());
        exit(1);
}
```

**JAVA API**

```
LBMSourceAttributes sattr = null;
try {
sattr = new LBMSourceAttributes();
sattr.setValue("ume_store", "127.0.0.1:14567:1000");
}
catch (LBMException ex) {
System.err.println("Error creating source attribute: " + ex.toString());
System.exit(1);
}
```

**.NET API**

```
LBMSourceAttributes sattr = null;
try {
sattr = new LBMSourceAttributes();
sattr.setValue("ume_store", "127.0.0.1:14567:1000");
}
catch (LBMException ex) {
System.Console.Error.WriteLine ("Error creating source attribute: " + ex.toString());
System.Environment.Exit(1);
       }
```

### *7.1.1.2. Save Assigned RegIDs*

Your application can save the RegID assigned to a source or receiver from the store because the **UMP** API informs your application of the RegID used for each registration. This method of managing RegIDs is perhaps the most flexible, but also requires some work by the application to save RegIDs and retrieve them in some way.

The following source code examples save the RegID assigned to a source to a file. (See also ume-example-src-3.c.)

**C API**

```
typedef struct src_info_t_stct {
int existing_regid;
int message_num;
} src_info_t;

#define SRC_REGID_SAVE_FILENAME "UME-example-src-RegID"

int save_src_regid_to_file(const char *filename, lbm_src_event_ume_registration_ex_t *reg)
{
FILE *fp;

if ((fp = fopen(filename, "w")) == NULL)
return -1;
fprintf(fp, "%s:%u", reg->store, reg->registration_id);
printf("saving RegID info to \"%s\" - %s:%u\n", filename, reg->store, reg->registration_id);
fflush(fp);
fclose(fp);
return 0;
}
```

### *7.1.1.3. Managing RegIDs with Session IDs*

The RegIDs used by stores to identify sources and receivers must be unique. Rather than maintaining RegIDs (either statically or dynamically), applications can use a Session ID, which is simply a 64-bit value that uniquely identifies any set of sources with unique topics and receivers with unique topics. A single Session ID allows **UMP** stores to correctly identify all the sources and receivers for a particular application.

Combinations of sources and receivers that make up a single valid session include the following.

- Sources for topics A, B, and C
- Receivers for topics A, B, and C
- Sources for topics A, B, and C, and receivers for topics X, Y and Z
- Sources for topics A, B, and C, and receivers for topics A, B, and C

> **Note:** Note that any topic can be used for a source and a receiver at the same time, but not for more than one of each. Two sources using topic A, for example, would need to be split into two different contexts.

The **UMP** configuration option, `ume_session_id` (../Config/ultramessagingpersistenceoptions.html#CONTEXTUMESESSIONID), specifies a Session ID for a source, receiver or a context. If you want all sources and receivers for a particular context to use the same Session ID, use `(context) ume_session_id` (../Config/ultramessagingpersistenceoptions.html#CONTEXTUMESESSIONID). Any source or receiver that does not specify its own Session ID inherits the context's session ID. If a source or receiver specifies its own Session ID, it overrides the context Session ID for that individual source or receiver.

Of the two mutually exclusive methods for managing RegIDs, ...

1. Enable your application to assign and manage every RegID, ensuring no two objects registered with an individual store share the same RegID.
2. Allow the store to assign every RegID and enable your application to persist the RegIDs.

... using Session IDs simplifies the second management method. Since you cannot combine these two strategies at any single store, you also cannot combine the first method with the use of Session IDs at a single store.

##### *7.1.1.3.1. How Stores Associate Session IDs and RegIDs*

Session IDs do not replace the use of RegIDs by **UMP** but rather simplify RegID management. Using Session IDs equates to your application specifying a 0 (zero) RegID for all sources and receivers. However, instead of your application persisting the RegID assigned by the store, the store maintains the RegID for you.

When a store receives a registration request from a source or receiver with a particular Session ID, it checks to see if it already has a source or receiver for that topic/Session ID. If it does, then it responds with that source's or receiver's RegID.

If it does not find a source or receiver for that topic/Session ID pair, the store ...

1. Assigns a new RegID.
2. Associates the topic/Session ID with the new RegID.
3. Responds to the source or receiver with the new RegID.

The source can then advertise with the RegID supplied by the store. Receivers include the source's RegID in their registration request.

### 7.1.2. UMP Sources

The major concerns of sources revolve around RegID management and message retention. This section discusses the following topics.

- *New or Re-Registration*
- *Sources Must Be Able to Resume Sending*
- *Source Message Retention and Release*
- *Source Release Policy Options*
- *Confirmed Delivery*

- *Sources Using Round-Robin Store Configuration*
- *Sources Using Quorum/Consensus Store Configuration*
- *Source Event Handler*
- *Source Event Handler - Stability, Confirmation and Release*
- *Mapping Your Message Numbers to UMS/UMP Sequence Numbers*
- *Receiver Liveness Detection*

#### *7.1.2.1. New or Re-Registration*

Any source needs to know at start-up if it is a new registration or a re-registration. The answer determines how a source registers with the store. **UMP** can not answer this question. Therefore, it is essential that the developer consider what identifies the lifetime of a source and how a source determines the appropriate value to use as the RegID when it is ready to register. RegIDs are per source per topic per store, thus a single RegID per store is needed.

The following source code examples look for an existing RegID from a file and uses a new RegID assigned from the store if it finds no existing RegID. (See also ume-example-src-3.c.)

**C API**

```
    err = lbm_context_create(&ctx, NULL, NULL, NULL);
    if (err) {printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);}

srcinfo.message_num = 1;
srcinfo.existing_regid = 0;

err = read_src_regid_from_file(SRC_REGID_SAVE_FILENAME, store_info, sizeof(store_info));
if (!err) { srcinfo.existing_regid = 1; }

        err = lbm_src_topic_attr_create(&attr);
        if (err) {printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);}

        err = lbm_src_topic_attr_str_setopt(attr, "ume_store", store_info);
        if (err) {printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);}
```

The use of Session IDs allows **UMP** , as opposed to your application, to accomplish the same RegID management. See *Managing RegIDs with Session IDs*.

#### *7.1.2.2. Sources Must Be Able to Resume Sending*

A source sends messages unless **UMP** prevents it, in which case, the send function returns an error. A source may lose the ability to send messages temporarily if the store(s) in use become unresponsive, e.g. the store(s) die or become disconnected from the source. Once the store(s) are responsive again, sending can continue. Thus source applications need to take into account that sending may fail temporarily under specific failure cases and be able to resume sending when the failure is removed.

The following source code examples demonstrate how a failed send function can sleep for a second and try again.

**C API**

```
while (lbm_src_send(src, message, len, 0) == LBM_FAILURE) {
```

```
If (lbm_errnum() == LBM_EUMENOREG) {
printf("Send unsuccessful. Waiting...\n");
sleep(1);
continue;
}
fprintf(stderr, "lbm_src_send: %s\n", lbm_errmsg());
                     exit(1);
                }
```

**JAVA API**

```
for (;;) {
try {
src.send(message, len, 0);
}
catch (UMENoRegException ex) {
System.out.println("Send unsuccessful. Waiting...");
try {
Thread.sleep(1000);
}
catch (InterruptedException e) { }
continue;
}
catch (LBMException ex) {
System.err.println("Error sending message: " + ex.toString());
System.exit(1);
}
break;
}
```

**.NET API**

```
for (;;) {
try {
src.send(message, len, 0);
}
catch (UMENoRegException ex) {
System.Console.Out.WriteLine("Send unsuccessful. Waiting...");
System.Threading.Thread.Sleep(1000);
continue;
}
catch (LBMException ex) {
System.Console.Out.WriteLine ("Error sending message: " + ex.toString());
System.exit(1);
}
break;
}
```

#### *7.1.2.3. Source Message Retention and Release*

**UMP** allows streaming of messages from a source without regard to message stability at a store, which is one reason for **UMP**'s performance advantage over other persistent messaging systems. Sources retain all messages until notified by the active store(s) that they are stable. This provides a method for stores to be brought up to date when restarted or started anew.

> **Note:** Source message retention is separate from the persistence of messages in the store.

When messages are considered stable at the store, the source can release them which frees up source retention memory for new messages. Generally, the source releases older stable messages first. To release the oldest retained message, all the following conditions must be met:

- message must meet stability requirements of the source, which can range from a single stability notice from the active store to stability notices from a group of stores (See *Sources Using Quorum/Consensus Store Configuration*)

  and

- message must have been confirmed as delivered by a configured number of receivers (`ume_retention_unique_confirmations`),

  and

- the aggregate amount of buffered messages exceeds `retransmit_retention_size_threshold` bytes in payload and headers.

Some things to note:

- If the `retransmit_retention_size_threshold` is not met, no messages will be released regardless of stability.
- If the source registered with a "no-cache" store (See *UMP Stores*) or `ume_message_stability_notification` is turned off, `ume_retention_unique_confirmations` is the only way to allow the source to release messages before retention size options come into play.
- If the aggregate amount of buffered messages exceeds `retransmit_retention_size_limit` bytes in payload and headers, then the oldest retained message is forcibly released even if it does not meet one or more of the conditions above. This condition should be avoided and suggests increasing the `retransmit_retention_size_limit` or lowering the `retransmit_retention_size_threshold`.

#### *7.1.2.4. Source Release Policy Options*

sources use a set of configuration options to release messages that, in effect, specify the source's release policy. The following configuration options directly impact when the source may release retained messages.

- `ume_message_stability_notification` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEMESSAGESTABILITYNOTIFICATION)
- `ume_retention_unique_confirmations` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERETENTIONUNIQUECONFIRMATIONS)

- `retransmit_retention_size_threshold` (../Config/latejoinoptions.html#SOURCERETRANSMITRETENTIONSIZETHRESHOLD)
- `retransmit_retention_size_limit` (../Config/latejoinoptions.html#SOURCERETRANSMITRETENTIONSIZELIMIT)

#### 7.1.2.5. Confirmed Delivery

As mentioned earlier, `ume_retention_unique_confirmations` requires a message to have a minimum number of unique confirmations from different receivers before the message may be released. This retains messages that have not been confirmed as being received and processed and keeps them available to fulfill any retransmission requests.

The following code samples show how to require a message to have 10 unique receiver confirmations

**C API**

```
lbm_src_topic_attr_t * sattr;

if (lbm_src_topic_attr_create(&sattr) == LBM_FAILURE) {
        fprintf(stderr, "lbm_src_topic_attr_create: %s\n", lbm_errmsg());
        exit(1);
}
if (lbm_src_topic_attr_str_setopt(sattr, "ume_retention_unique_confirmations",
"10")
== LBM_FAILURE) {
        fprintf(stderr, "lbm_src_topic_attr_str_setopt: %s\n", lbm_errmsg());
        exit(1);
}
```

**JAVA API**

```
LBMSourceAttributes sattr = null;
try {
sattr = new LBMSourceAttributes();
sattr.setValue("ume_retention_unique_confirmations", "10");
}
catch (LBMException ex) {
System.err.println("Error creating source attribute: " + ex.toString());
System.exit(1);
}
```

**.NET API**

```
LBMSourceAttributes sattr = null;
try {
sattr = new LBMSourceAttributes();
sattr.setValue("ume_retention_unique_confirmations", "10");
}
catch (LBMException ex) {
System.Console.Error.WriteLine ("Error creating source attribute: " + ex.toString());
System.Environment.Exit(1);
        }
```

#### *7.1.2.6. Sources Using Round-Robin Store Configuration*

The source retains messages until they are considered stable at the active store(s). For Round-Robin store behavior, this means the current active store notifies the source that it has stabilized the message via a message stability notification. The following configuration file statements implement Round-Robin behavior among 3 stores.

```
source ume_store 10.29.3.77:15313:150000:0
source ume_store 10.29.3.76:16313:160000:0
source ume_store 10.29.3.75:17313:170000:0
source ume_message_stability_notification 1
source ume_store_behavior rr
```

See also *Round-Robin Store Usage*

#### *7.1.2.7. Sources Using Quorum/Consensus Store Configuration*

In the case of Quorum/Consensus store behavior, a message is considered stable after it has been successfully stored within a group of stores or among groups of stores according to the two settings, intergroup behavior and intragroup behavior, described below.

- The intragroup behavior (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERETENTIONINTRAGROUPSTABILITYBEHAVIOR) specifies the requirements needed to stabilize a message among the stores within a group. A message is stable for the group once it is successfully stored at a quorum (majority) of the group's stores or successfully stored in all the stores in the group.
- The intergroup behavior (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERETENTIONINTERGROUPSTABILITYBEHAVIOR) specifies the requirements needed to stabilize a message among groups of stores. A message is stable among the groups if it is successfully stored at any group, a majority of groups, or all groups.

Notice that a message needs to meet intragroup stability requirements before it can meet intergroup stability requirements. These options provide a number of possibilities for retention of messages for the source.

The following configuration file statements implement a 3-group Quorum/Consensus configuration with each group on a different machine, in which a message is considered stable when it has been successfully stored at a quorum of stores in at least one group. (See *Quorum/Consensus - Single Location Groups* for more information about this configuration.)

```
source ume_store 10.29.3.77:10313:101000:0
source ume_store 10.29.3.77:11313:110000:0
source ume_store 10.29.3.77:12313:120000:0
source ume_store 10.29.3.77:13313:130000:0
source ume_store 10.29.3.77:14313:140000:0
source ume_store 10.29.3.78:15313:150000:1
source ume_store 10.29.3.78:16313:160000:1
source ume_store 10.29.3.78:17313:170000:1
source ume_store 10.29.3.79:18313:180000:2
```

```
source ume_store 10.29.3.79:19313:190000:2
source ume_store 10.29.3.79:29313:290000:2
source ume_store 10.29.3.79:39313:390000:2
source ume_store 10.29.3.79:49313:490000:2

source ume_message_stability_notification 1
source ume_store_behavior qc

source ume_store_group 0:5
source ume_store_group 1:3
source ume_store_group 2:5

source ume_retention_intragroup_stability_behavior quorum
source ume_retention_intergroup_stability_behavior any
```

See also *Quorum/Consensus Store Usage* and *Quorum/Consensus - Mixed Location Groups*.

#### *7.1.2.8. Source Event Handler*

The Source Event Handler is a function callback initialized at source creation to provide source events to your application related to the operation of the source. The following source code examples illustrate the use of a source event handler for registration events. To accept other source events, additional case statements would be required, one for each additional source event. See also *UMP and UMQ Events*.

**C API**

```
int handle_src_event(lbm_src_t *src, int event, void *ed, void *cd)
{
    switch (event) {
    case LBM_SRC_EVENT_UME_REGISTRATION_ERROR:
{
const char *errstr = (const char *)ed;
      printf("Error registering source with UME store: %s\n", errstr);
}
break;
        case LBM_SRC_EVENT_UME_REGISTRATION_SUCCESS_EX:
         {
          lbm_src_event_ume_registration_ex_t *reg =
                        (lbm_src_event_ume_registration_ex_t *)ed;

             printf("UME store %u: %s registration success. RegID %u. Flags %x ",
                               reg->store_index, reg->store, reg->registration_id,
                               reg->flags);
             if (reg->flags & LBM_SRC_EVENT_UME_REGISTRATION_SUCCESS_EX_FLAG_OLD)
                 printf("OLD[SQN %x] ", reg->sequence_number);
             if (reg->flags & LBM_SRC_EVENT_UME_REGISTRATION_SUCCESS_EX_FLAG_NOACKS)
                 printf("NOACKS ");
             printf("\n");
         }
         break;
        case LBM_SRC_EVENT_UME_REGISTRATION_COMPLETE_EX:
         {
```

```
            lbm_src_event_ume_registration_complete_ex_t *reg;

                        reg  = (lbm_src_event_ume__complete_ex_t *)ed;
            printf("UME registration complete. SQN %x. Flags %x ", reg->sequence_number,
                        reg->flags);
            if (reg->flags & LBM_SRC_EVENT_UME_REGISTRATION_COMPLETE_EX_FLAG_QUORUM)
                printf("QUORUM ");
            printf("\n");
        }
        break;
       case LBM_SRC_EVENT_UME_STORE_UNRESPONSIVE:
       {
           const char *infostr = (const char *)ed;
           printf("UME store: %s\n", infostr);
       }
       break;
    default:
         printf("Unknown source event %d\n", event);
         break;
    }
   return 0;
}
```

**JAVA API**

```
public int onSourceEvent(Object arg, LBMSourceEvent sourceEvent)
{
    switch (sourceEvent.type()) {
    case LBM.SRC_EVENT_UME_REGISTRATION_ERROR:
System.out.println("Error registering source with UME store: "
+ sourceEvent.dataString());
break;
    case LBM.SRC_EVENT_UME_REGISTRATION_SUCCESS_EX:
 UMESourceEventRegistrationSuccessInfo reg = sourceEvent.registrationSuccessInfo();
 System.out.print("UME store " + reg.storeIndex() + ": " + reg.store()
+ " registration success. RegID " + reg.registrationId() + ". Flags "
+ reg.flags() + " ");
                if (((reg.flags() & LBM.SRC_EVENT_UME_REGISTRATION_SUCCESS_EX_FLAG_OLD))
                != 0) {
 System.out.print("OLD[SQN " + reg.sequenceNumber() + "] ");
                }
                if (((reg.flags() & LBM.SRC_EVENT_UME_REGISTRATION_SUCCESS_EX_FLAG_NOACKS))
                != 0) {
System.out.print("NOACKS ");
                }
                System.out.println();
                break;
    case LBM.SRC_EVENT_UME_REGISTRATION_COMPLETE_EX:
                UMESourceEventRegistrationCompleteInfo regcomp =
                sourceEvent.registrationCompleteInfo();
                System.out.print("UME registration complete. SQN " + regcomp.sequenceNumber()
+ ". Flags " + regcomp.flags() + " ");
```

```
                if ((regcomp.flags() &
                LBM.SRC_EVENT_UME_REGISTRATION_COMPLETE_EX_FLAG_QUORUM) != 0) {
System.out.print("QUORUM ");
                }
                System.out.println();
                break;
    case LBM.SRC_EVENT_UME_STORE_UNRESPONSIVE:
                System.out.println("UME store: "
                    + sourceEvent.dataString());
                break;
    ...
    default:
                System.out.println("Unknown source event "
+ sourceEvent.type());
                break;
        }
        return 0;
}
```

### .NET API

```
public int onSourceEvent(Object arg, LBMSourceEvent sourceEvent)
{
    switch (sourceEvent.type()) {
    case LBM.SRC_EVENT_UME_REGISTRATION_ERROR:
System.Console.Out.WriteLine("Error registering source with UME store: "
+ sourceEvent.dataString());
break;
    case LBM.SRC_EVENT_UME_REGISTRATION_SUCCESS_EX:
 UMESourceEventRegistrationSuccessInfo reg = sourceEvent.registrationSuccessInfo();
 System.Console.Out.Write("UME store " + reg.storeIndex() + ": " + reg.store()
+ " registration success. RegID " + reg.registrationId() + ". Flags "
+ reg.flags() + " ");
                if (((reg.flags() & LBM.SRC_EVENT_UME_REGISTRATION_SUCCESS_EX_FLAG_OLD))
                != 0) {
 System.Console.Out.Write("OLD[SQN " + reg.sequenceNumber() + "] ");
                }
                if (((reg.flags() & LBM.SRC_EVENT_UME_REGISTRATION_SUCCESS_EX_FLAG_NOACKS))
                != 0) {
System.Console.Out.Write("NOACKS ");
                }
                System.Console.Out.WriteLine();
                break;
    case LBM.SRC_EVENT_UME_REGISTRATION_COMPLETE_EX:
                UMESourceEventRegistrationCompleteInfo regcomp =
                sourceEvent.registrationCompleteInfo();
                System.Console.Out.Write("UME registration complete. SQN " +
                regcomp.sequenceNumber()
+ ". Flags " + regcomp.flags() + " ");
                if ((regcomp.flags() &
                LBM.SRC_EVENT_UME_REGISTRATION_COMPLETE_EX_FLAG_QUORUM) != 0) {
System.Console.Out.Write("QUORUM ");
```

```
                }
                System.Console.Out.WriteLine();
                break;
    case LBM.SRC_EVENT_UME_STORE_UNRESPONSIVE:
                System.Console.Out.WriteLine("UME store: "
                        + sourceEvent.dataString());
                break;
    ...
    default:
                System.Console.Out.WriteLine("Unknown source event "
+ sourceEvent.type());
                break;
        }
        return 0;
}
```

#### *7.1.2.9. Source Event Handler - Stability, Confirmation and Release*

As shown in Section 7.1.2.8 above, the Source Event Handler can be expanded to handle more source events by adding additional case statements. The following source code examples show case statements to handle message stability events, delivery confirmation events and message release (reclaim) events. See also *UMP and UMQ Events*.

**C API**

```
case LBM_SRC_EVENT_UME_MESSAGE_STABLE_EX:
/* requires that source ume_message_stability_notification attribute is enabled */
        {
            lbm_src_event_ume_ack_ex_info_t *info = (lbm_src_event_ume_ack_ex_info_t *)ed;

            printf("UME store %u: %s message stable. SQN %x (msgno %d). Flags %x ",
            info->store_index, info->store,
                    info->sequence_number, (int)info->msg_clientd - 1, info->flags);
            if (info->flags & LBM_SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_INTRAGROUP_STABLE)
                    printf("IA "); /* Stable within store group */
            if (info->flags & LBM_SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_INTERGROUP_STABLE)
                    printf("IR "); /* Stable amongst all stores */
            if (info->flags & LBM_SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_STABLE)
                    printf("STABLE ");  /* Just plain stable */
            if (info->flags & LBM_SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_STORE)
                    printf("STORE ");   /* Stability reported by UME Store */
            printf("\n");
        }
        break;

case LBM_SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX:
/* requires that source ume_confirmed_delivery_notification attribute is enabled */
        {
            lbm_src_event_ume_ack_ex_info_t *info = (lbm_src_event_ume_ack_ex_info_t *)ed;

             printf("UME delivery confirmation. SQN %x, Receiver RegID %u (msgno %d). Flags %x ",
                      info->sequence_number, info->rcv_registration_id,
```

```
                    (int)info->msg_clientd - 1, info->flags);
                if (info->flags & LBM_SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_UNIQUEACKS)
                    printf("UNIQUEACKS ");
                    /* Satisfied number of unique ACKs requirement */
                if (info->flags & LBM_SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_UREGID)
                    printf("UREGID ");
                    /* Confirmation contains receiver application registration ID */
                if (info->flags & LBM_SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_OOD)
                    printf("OOD ");
                    /* Confirmation received from arrival order receiver */
                if (info->flags & LBM_SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_EXACK)
                    printf("EXACK ");
                    /* Confirmation explicitly sent by receiver */
                printf("\n");
        }
        break;

case LBM_SRC_EVENT_UME_MESSAGE_RECLAIMED:
/* requires that source ume_confirmed_delivery_notification or ume_message_stability_notification
attributes are enabled */
        {
                lbm_src_event_ume_ack_info_t *ackinfo = (lbm_src_event_ume_ack_info_t *)ed;

                printf("UME message released - sequence number %x (msgno %d)\n",
                        ackinfo->sequence_number, (int)ackinfo->msg_clientd - 1);
        }
        break;
```

**JAVA API**

```
case LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX:
// requires that source ume_message_stability_notification attribute is enabled
        UMESourceEventAckInfo staInfo = sourceEvent.ackInfo();
        System.out.print("UME store " + staInfo.storeIndex() + ": "
                    + staInfo.store() + " message stable. SQN " + staInfo.sequenceNumber()
                    + " (msgno " + staInfo.clientObject() + "). Flags "
                    + staInfo.flags() + " ");
        if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_INTRAGROUP_STABLE)
        != 0) {
                System.out.print("IA "); // Stable within store group
        }
        if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_INTERGROUP_STABLE)
        != 0) {
                System.out.print("IR ");  // Stable amongst all stores
        }
        if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_STABLE) != 0) {
                System.out.print("STABLE ");  // Just plain stable
        }
        if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_STORE) != 0) {
                System.out.print("STORE ");   // Stability reported by UME Store
        }
```

```
        System.out.println();
        break;

case LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX:
// requires that source ume_confirmed_delivery_notification attribute is enabled
        UMESourceEventAckInfo cdelvinfo = sourceEvent.ackInfo();
        System.out.print("UME delivery confirmation. SQN " + cdelvinfo.sequenceNumber()
                    + ", RcvRegID " + cdelvinfo.receiverRegistrationId() + " (msgno "
                    + cdelvinfo.clientObject() + "). Flags " + cdelvinfo.flags() + " ");
        if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_UNIQUEACKS)
        != 0) {
                System.out.print("UNIQUEACKS "); // Satisfied number of unique ACKs requirement
        }
        if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_UREGID)
        != 0) {
                System.out.print("UREGID ");    // Confirmation contains receiver application
                registration ID
        }
        if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_OOD)
        != 0) {
                System.out.print("OOD ");      // Confirmation received from arrival order
                receiver
        }
        if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_EXACK)
        != 0) {
                System.out.print("EXACK ");    // Confirmation explicitly sent by receiver
        }
        System.out.println();
        break;

case LBM.SRC_EVENT_UME_MESSAGE_RECLAIMED:
// requires that source ume_confirmed_delivery_notification or
// ume_message_stability_notification attributes are enabled
        System.out.println("UME message released - sequence number "
                + Long.toHexString(sourceEvent.sequenceNumber())
                + " (msgno "
                + Long.toHexString(((Integer)sourceEvent.clientObject()).longValue())
                + ")");
        break;
```

### .NET API

```
case LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX:
// requires that source ume_message_stability_notification attribute is enabled
            UMESourceEventAckInfo staInfo = sourceEvent.ackInfo();
            System.Console.Out.Write("UME store " + staInfo.storeIndex() + ": "
                        + staInfo.store() + " message stable. SQN " + staInfo.sequenceNumber()
                        + " (msgno " + ((int)staInfo.clientObject()).ToString("x") + ").
                        Flags " + staInfo.flags() + " ");
            if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_INTRAGROUP_STABLE)
            != 0)
            {
```

```
                System.Console.Out.Write("IA ");  // Stable within store group
            }
            if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_INTERGROUP_STABLE)
            != 0)
            {
                System.Console.Out.Write("IR ");  // Stable amongst all stores
            }
            if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_STABLE) != 0)
            {
                System.Console.Out.Write("STABLE ");  // Just plain stable
            }
            if ((staInfo.flags() & LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX_FLAG_STORE) != 0)
            {
                System.Console.Out.Write("STORE ");  // Stability reported by UME Store
            }
            System.Console.Out.WriteLine();
            break;

case LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX:
// requires that source ume_confirmed_delivery_notification attribute is enabled

            UMESourceEventAckInfo cdelvinfo = sourceEvent.ackInfo();

            System.Console.Out.Write("UME delivery confirmation. SQN " +
            cdelvinfo.sequenceNumber()
                        + ", RcvRegID " + cdelvinfo.receiverRegistrationId() + " (msgno "
                        + ((int)cdelvinfo.clientObject()).ToString("x") + "). Flags " +
                        cdelvinfo.flags() + " ");
            if ((cdelvinfo.flags() &
            LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_UNIQUEACKS) != 0)
            {
                System.Console.Out.Write("UNIQUEACKS ");  // Satisfied number of unique
                ACKs requirement
            }
            if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_UREGID)
            != 0)
            {
                System.Console.Out.Write("UREGID ");  // Confirmation contains receiver
                application registration ID
            }
            if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_OOD)
            != 0)
            {
                System.Console.Out.Write("OOD ");  // Confirmation received from arrival
                order receiver
            }
            if ((cdelvinfo.flags() & LBM.SRC_EVENT_UME_DELIVERY_CONFIRMATION_EX_FLAG_EXACK)
            != 0)
            {
                System.Console.Out.Write("EXACK ");  // Confirmation explicitly sent by
                receiver
            }
            System.Console.Out.WriteLine();
```

```
            break;

case LBM.SRC_EVENT_UME_MESSAGE_RECLAIMED:
// requires that source ume_confirmed_delivery_notification or
// ume_message_stability_notification attributes are enabled

            System.Console.Out.WriteLine("UME message released - sequence number "
                                   + sourceEvent.sequenceNumber().ToString("x")
                                   + " (msgno "
                                   + ((int)sourceEvent.clientObject()).ToString("x")
                                   + ")");
            break;
```

### 7.1.2.10. Mapping Your Message Numbers to *UMS*/*UMP* Sequence Numbers

`lbm_src_sendv_ex()` allows you to create a pointer to an object or structure. This pointer will be returned to your application along with all source events. You can then update the object or structure with source event information. For example, if your messages exceed 8K - which requires fragmentation your application's message into more than one **UM** message - receiving sequence number events with this pointer allows you to determine all the **UM** sequence numbers for the message and, therefore, how many release (reclaim) events to expect. The following two source code examples show how to:

- Enable message sequence number information
- Handle sequence number source events to determine the application message number in the Source Event Handler

**C API - Enable Message Information**

```
lbm_src_send_ex_info_t exinfo;

/* Enable message sequence number info to be returned */
exinfo.flags = LBM_SRC_SEND_EX_FLAG_UME_CLIENTD | LBM_SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO;
exinfo.ume_msg_clientd = (void *)(msgno + 1);
/* msgno set to application message number (can't evaluate to NULL) */
while (lbm_src_send_ex(src, message, msglen, 0, &exinfo) == LBM_FAILURE)
{
if (lbm_errnum() == LBM_EUMENOREG)
        {
printf("Send unsuccessful. Waiting...\n");
                SLEEP_MSEC(1000); /* Sleep for 1 second */
        }
else
{
  fprintf(stderr, "lbm_src_send: %s\n", lbm_errmsg());
break;
}
}
```

**C API - Sequence Number Event Handler**

```
int handle_src_event(lbm_src_t *src, int event, void *ed, void *cd)
{
      switch (event) {
      case LBM_SRC_EVENT_SEQUENCE_NUMBER_INFO:
            {
                  lbm_src_event_sequence_number_info_t *info =
                  (lbm_src_event_sequence_number_info_t *)ed;

                  if (info->first_sequence_number != info->last_sequence_number) {
                        printf("SQN [%x,%x] (msgno %d)\n", info->first_sequence_number,
                        info->last_sequence_number, (int)info->msg_clientd - 1);
                  } else {
                        printf("SQN %x (msgno %d)\n", info->last_sequence_number,
                        (int)info->msg_clientd - 1);
                }
            }
            break;
      ...
      }
      return 0;
}
```

### JAVA API - Enable Message Information

```
LBMSourceSendExInfo exinfo = new LBMSourceSendExInfo();
exinfo.setClientObject(new Integer(msgno));  // msgno set to application message number
exinfo.setFlags(LBM.SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO);
// Enable message sequence number info to be returned
for (;;)
{
try
{
src.send(message, msglen, 0, exinfo);
}
catch(UMENoRegException ex)
{
         try
         {
         Thread.sleep(1000);
         }
         catch (InterruptedException e) { }
continue;

}
catch (LBMException ex)
{
         System.err.println("Error sending message: " + ex.toString());
}
break;
}
```

### JAVA API - Sequence Number Event Handler

```
public int onSourceEvent(Object arg, LBMSourceEvent sourceEvent)
{
      switch (sourceEvent.type())
      {
            case LBM.SRC_EVENT_SEQUENCE_NUMBER_INFO:
                  LBMSourceEventSequenceNumberInfo info = sourceEvent.sequenceNumberInfo();
                  if (info.firstSequenceNumber() != info.lastSequenceNumber()) {
                        System.out.println("SQN [" + info.firstSequenceNumber()
                            + "," + info.lastSequenceNumber() + "] (msgno "
                            + info.clientObject() + ")");
                  }
                  else {
                        System.out.println("SQN " + info.lastSequenceNumber()
                            + " (msgno " + info.clientObject() + ")");
                  }
                  break;
          ...
      }
      return 0;
}
```

### .NET API - Enable Message Information

```
LBMSourceSendExInfo exinfo = new LBMSourceSendExInfo();
exinfo.setClientObject(msgno);  // msgno set to application message number
exinfo.setFlags(LBM.SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO);
// Enable message sequence number info to be returned
for (;;)
{
try
{
src.send(message, msglen, 0, exinfo);
}
catch(UMENoRegException ex)
{
         System.Threading.Thread.Sleep(100);
continue;

}
catch (LBMException ex)
{
         System.Console.Out.WriteLine("Error sending message: " + ex.Message());
}
break;
}
```

### .NET API - Sequence Number Event Handler

```
public void onSourceEvent(Object arg, LBMSourceEvent sourceEvent)
```

```
{
	switch (sourceEvent.type())
	{
		case LBM.SRC_EVENT_SEQUENCE_NUMBER_INFO:
			LBMSourceEventSequenceNumberInfo info = sourceEvent.sequenceNumberInfo();
			if (info.firstSequenceNumber() != info.lastSequenceNumber())
			{
				System.Console.Out.WriteLine("SQN [" + info.firstSequenceNumber()
					+ "," + info.lastSequenceNumber() + "] (cd "
					+ ((int)info.clientObject()).ToString("x") + ")");
			}
			else
			{
				System.Console.Out.WriteLine("SQN " + info.lastSequenceNumber()
					+ " (msgno " + ((int)info.clientObject()).ToString("x") + ")");
			}
			break;
		...

	}
	return 0;
}
```

#### *7.1.2.11. Receiver Liveness Detection*

As an extension to *Confirmed Delivery*, you can set receivers to send a keepalive to a source during a measured absence of delivery confirmations (due to traffic lapse). In the event that neither message reaches the source within a designated interval, or if the delivery confirmation TCP connection breaks down, the receiver is assumed to have "died". **UM** then notifies the publishing application via context event callback. This lets the publisher assign a new subscriber.

To use this feature, set these five configuration options:

- `ume_source_liveness_timeout` (../Config/ultramessagingpersistenceoptions.html#UMESOURCELIVENESSTIMEOUT)
- `ume_receiver_liveness_interval` (../Config/ultramessagingpersistenceoptions.html#UMERECEIVERLIVENESSINTERVAL)
- `ume_confirmed_delivery_notification` (../Config/ultramessagingpersistenceoptions.html#UMECONFIRMEDDELIVERYNOTIFICATION)
- `ume_user_receiver_registration_id` (../Config/ultramessagingpersistenceoptions.html#UMEUSERRECEIVERREGISTRATIONID)
- `ume_session_id` (../Config/ultramessagingpersistenceoptions.html#UMESESSIONID)

> **Note:** You must set the `ume_source_liveness_timeout` option to 5 times the value of `ume_receiver_liveness_interval`.

This specialized feature is not recommended for general use. If you are considering it, please note the following caveats:

- Do not use in conjunction with a **UM** Gateway.
- There is a variety of potential network occurrences that can break or reset the TCP connection and falsely indicate the death of a receiver.
- In cases where a receiver object is deleted while its context is not, the publisher may still falsely assume the receiver to be alive. Other false receiver-alive assumptions could be caused by the following:
- TCP connections can enter a half-open or otherwise corrupted state.
- Failed TCP connections sometimes do not fully close, or experience objectionable delays before fully closing.
- A switch or router failure along the path does not affect the TCP connection state.

## 7.1.3. UMP Receivers

Receivers are predominantly interested in RegID management and recovery management. This section discusses the following topics.

- *Receiver RegID Management*
- *Receiver Message and Event Handler*
- *Recovery Management*
- *Setting Callback Function to Set Recovery Sequence Number*
- *Message Consumption*

### *7.1.3.1. Receiver RegID Management*

RegIDs are slightly more involved for receivers than for sources. Since RegIDs are per source per topic per store and a topic may have several sources, a receiver may have to manage several RegIDs per store in use. Fortunately, receivers in **UMP** can leverage the RegID of the source with the use of a callback as discussed in *Adding Fault Recovery with Registration IDs* and shown in ume-example-rcv-2.c. Your application can determine the correct RegID to use and return it to **UMP** . You can also use Session IDs to enable **UMP** to manage receiver RegIDs. See *Managing RegIDs with Session IDs*.

Much like sources, receivers typically have a lifetime based on an amount of work, perhaps an infinite amount. And just like sources, it may be helpful to consider that a RegID is "assigned" at the start of that work and is out of use at the end. In between, the RegID is in use by the instance of the receiver application. However, the nature of RegIDs being per source means that the expected lifetime of a source should play a role in how RegIDs on the receiver are managed. Thus, it may be helpful for the application developer to consider the source application lifetime when deciding how best to handle RegIDs on the receiver.

### *7.1.3.2. Receiver Message and Event Handler*

The Receiver Message and Event Handler is a function callback started at receiver initialization to provide Receiver messages to your application on behalf of the receiver. The following source code examples illustrate the use of a

receiver message and event handler for registration messages. To accept other receiver events, additional case statements would be required, one for each additional event. See also *UMP and UMQ Events*.

**C API**

```
int rcv_handle_msg(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)
{
    switch (msg->type) {
    ...
    case LBM_MSG_UME_REGISTRATION_ERROR:
         printf("[%s][%s] UME registration error: %s\n", msg->topic_name, msg->source,
                msg->data);
         exit(0);
    break;
    case LBM_MSG_UME_REGISTRATION_SUCCESS:
         {
             lbm_msg_ume_registration_t *reg = (lbm_msg_ume_registration_t *)
                        (msg->data);

             printf("[%s][%s] UME registration successful. SrcRegID %u RcvRegID %u\n",
                    msg->topic_name, msg->source, reg->src_registration_id,
                         reg->rcv_registration_id);
         }
         break;
    case LBM_MSG_UME_REGISTRATION_SUCCESS_EX:
         {
             lbm_msg_ume_registration_ex_t *reg = (lbm_msg_ume_registration_ex_t *)
                        (msg->data);

             printf("[%s][%s] store %u: %s UME registration successful. SrcRegID %u
                        RcvRegID %u. Flags %x ",
                msg->topic_name, msg->source, reg->store_index, reg->store,
                reg->src_registration_id, reg->rcv_registration_id, reg->flags);
              if (reg->flags & LBM_MSG_UME_REGISTRATION_SUCCESS_EX_FLAG_OLD)
                 printf("OLD[SQN %x] ", reg->sequence_number);
             if (reg->flags & LBM_MSG_UME_REGISTRATION_SUCCESS_EX_FLAG_NOCACHE)
                 printf("NOCACHE ");
             printf("\n");
         }
         break;
    case LBM_MSG_UME_REGISTRATION_COMPLETE_EX:
         {
             lbm_msg_ume_registration_complete_ex_t *reg;

reg  = (lbm_msg_ume_registration_complete_ex_t *)(msg->data);
printf("[%s][%s] UME registration complete. SQN %x. Flags %x ",
                msg->topic_name, msg->source, reg->sequence_number, reg->flags);
             if (reg->flags & LBM_MSG_UME_REGISTRATION_COMPLETE_EX_FLAG_QUORUM)
                 printf("QUORUM ");
             if (reg->flags & LBM_MSG_UME_REGISTRATION_COMPLETE_EX_FLAG_RXREQMAX)
                 printf("RXREQMAX ");
             printf("\n");
         }
```

```
        break;
    case LBM_MSG_UME_REGISTRATION_CHANGE:
         printf("[%s][%s] UME registration change: %s\n", msg->topic_name, msg->source,
                msg->data);
         break;
    ...
    default:
         printf("Unknown lbm_msg_t type %x [%s][%s]\n", msg->type, msg->topic_name,
                msg->source);
         break;
    }
    return 0;
}
```

**JAVA API**

```
public int onReceive(Object cbArg, LBMMessage msg)
{
    case LBM.MSG_UME_REGISTRATION_ERROR:
            System.out.println("[" + msg.topicName() + "][" + msg.source()
                    + "] UME registration error: " + msg.dataString());
            break;
    case LBM.MSG_UME_REGISTRATION_SUCCESS_EX:
            UMERegistrationSuccessInfo reg = msg.registrationSuccessInfo();
            System.out.print("[" + msg.topicName() + "][" + msg.source()
                    + "] store " + reg.storeIndex() + ": "
                    + reg.store() + " UME registration successful. SrcRegID "
                    + reg.sourceRegistrationId() + " RcvRegID " + reg.receiverRegistrationId()
                    + ". Flags " + reg.flags() + " ");
            if ((reg.flags() & LBM.MSG_UME_REGISTRATION_SUCCESS_EX_FLAG_OLD) != 0)
                System.out.print("OLD[SQN " + reg.sequenceNumber() + "] ");
            if ((reg.flags() & LBM.MSG_UME_REGISTRATION_SUCCESS_EX_FLAG_NOCACHE) != 0)
                System.out.print("NOCACHE ");
            System.out.println();
            break;
    case LBM.MSG_UME_REGISTRATION_COMPLETE_EX:
            UMERegistrationCompleteInfo regcomplete = msg.registrationCompleteInfo();
            System.out.print("[" + msg.topicName() + "][" + msg.source()
                    + "] UME registration complete. SQN " + regcomplete.sequenceNumber()
                    + ". Flags " + regcomplete.flags() + " ");
            if ((regcomplete.flags() & LBM.MSG_UME_REGISTRATION_COMPLETE_EX_FLAG_QUORUM)
            != 0) {
                System.out.print("QUORUM ");
            }
            if ((regcomplete.flags() & LBM.MSG_UME_REGISTRATION_COMPLETE_EX_FLAG_RXREQMAX)
            != 0) {
                System.out.print("RXREQMAX ");
            }
            System.out.println();
            break;
    case LBM.MSG_UME_REGISTRATION_CHANGE:
            System.out.println("[" + msg.topicName() + "][" + msg.source()
```

```
                + "] UME registration change: " + msg.dataString());
            break;
    ...
    default:
            System.err.println("Unknown lbm_msg_t type " + msg.type() + " ["
                    + msg.topicName() + "][" + msg.source() + "]");
            break;
        }
    return 0;
}
```

### .NET API

```
public int onReceive(Object cbArg, LBMMessage msg)
{
    case LBM.MSG_UME_REGISTRATION_ERROR:
            System. Console.Out.WriteLine("[" + msg.topicName() + "][" + msg.source()
                    + "] UME registration error: " + msg.dataString());
            break;
    case LBM.MSG_UME_REGISTRATION_SUCCESS_EX:
            UMERegistrationSuccessInfo reg = msg.registrationSuccessInfo();
            System.Console.Out.Write("[" + msg.topicName() + "][" + msg.source()
                    + "] store " + reg.storeIndex() + ": "
                    + reg.store() + " UME registration successful. SrcRegID "
                    + reg.sourceRegistrationId() + " RcvRegID " + reg.receiverRegistrationId()
                    + ". Flags " + reg.flags() + " ");
            if ((reg.flags() & LBM.MSG_UME_REGISTRATION_SUCCESS_EX_FLAG_OLD) != 0)
                 System.Console.Out.Write ("OLD[SQN " + reg.sequenceNumber() + "] ");
            if ((reg.flags() & LBM.MSG_UME_REGISTRATION_SUCCESS_EX_FLAG_NOCACHE) != 0)
                 System.Console.Out.Write ("NOCACHE ");
            System.Console.Out.WriteLine();
            break;
    case LBM.MSG_UME_REGISTRATION_COMPLETE_EX:
            UMERegistrationCompleteInfo regcomplete = msg.registrationCompleteInfo();
            System.Console.Out.Write("[" + msg.topicName() + "][" + msg.source()
                    + "] UME registration complete. SQN " + regcomplete.sequenceNumber()
                    + ". Flags " + regcomplete.flags() + " ");
            if ((regcomplete.flags() & LBM.MSG_UME_REGISTRATION_COMPLETE_EX_FLAG_QUORUM)
            != 0) {
                System.Console.Out.Write("QUORUM ");
            }
            if ((regcomplete.flags() & LBM.MSG_UME_REGISTRATION_COMPLETE_EX_FLAG_RXREQMAX)
            != 0) {
                System.Console.Out.Write("RXREQMAX ");
            }
System.Console.Out.WriteLine();
            break;
    case LBM.MSG_UME_REGISTRATION_CHANGE:
            System.Console.Out.WriteLine("[" + msg.topicName() + "][" + msg.source()
                    + "] UME registration change: " + msg.dataString());
            break;
    ...
```

```
    default:
            System.Console.Out.WriteLine("Unknown lbm_msg_t type " + msg.type() + " ["
                    + msg.topicName() + "][" + msg.source() + "]");
            break;
        }
    return 0;
}
```

#### 7.1.3.3. Recovery Management

Recovery management for receivers is fairly simple because **UMP** requests any missing messages from the store(s) and delivers them as they are retransmitted. However, your application can specify a different message to begin retransmission with using either the `retransmit_request_maximum` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTMAXIMUM) configuration option or `lbm_ume_rcv_recovery_info_ex_func_t`.

For example, assume a source sends 7 messages with sequence numbers 0-6 which are stabilized at the store. The receiver, configured with the `retransmit_request_maximum` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTMAXIMUM) set to 2, consumes message 0, goes down, then comes back at message 6. `lbm_ume_rcv_recovery_info_ex_func_t` returns the following:

```
high_sequence_number = 6
low_rxreq_max_sequence_number = 4
low_sequence_number = 1

NOTE: low_rxreq_max_sequence_number = high_sequence_number - retransmit_request_maximum
```

- **UMP** obeys the `retransmit_request_maximum` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTMAXIMUM) configuration option and restarts with message 4. This is the default.
- If you modify the `low_sequence_number` to satisfy some other requirements, you can override the configuration option and restart at message 0, 2, 3, 5 or 6. See *Setting Callback Function to Set Recovery Sequence Number* below.
- The only way to restart at message 1 in this case, is to set the `retransmit_request_maximum` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTMAXIMUM) configuration option to its default value of 0. If your application changes the `low_sequence_number` and for whatever reason, the calculation results in the same value as the `low_sequence_number`, **UMP** ignores the calculation and restarts with message 4.

All messages retransmitted to a receiver are marked as retransmissions via a flag in the message structure. Thus it is easy for an application to determine if a message is a new message from the source or a retransmission, which may or may not have been processed before the failure. The presence or absence of the retransmit flag gives the application a hint of how best to handle the message with regard to it being processed previously or not.

#### *7.1.3.4. Setting Callback Function to Set Recovery Sequence Number*

The sample source code below demonstrates how to use the recovery sequence number info function to determine the stored message with which to restart a receiver. This method retrieves the low sequence number from the recovery sequence number structure and adds an offset to determine the beginning sequence number. The offset is a value completely under the control of your application. For example, if a receiver was down for a "long" period and you only want the receiver to receive the last 10 messages, use an offset to start the receiver with the 10th most recent message. If you wish not to receive any messages, set the `low_sequence_number` to the `high_sequence_number` plus one.

**C API**

```
lbm_ume_rcv_recovery_info_ex_func_t cb;

cb.func = ume_rcv_seqnum_ex;
cb.clientd = NULL;
if (lbm_rcv_topic_attr_setopt(&rcv_attr, "ume_recovery_sequence_number_info_function",
&cb, sizeof(cb)) == LBM_FAILURE) {
            fprintf(stderr,
            "lbm_rcv_topic_attr_setopt:ume_recovery_sequence_number_info_function: %s\n",
            lbm_errmsg());
            exit(1);
}
printf("Will use seqnum info with low offset %u.\n", seqnum_offset);

int ume_rcv_seqnum_ex(lbm_ume_rcv_recovery_info_ex_func_info_t *info, void *clientd)
{
     lbm_uint_t new_lo = info->low_sequence_number + seqnum_offset;

     printf("[%s] SQNs Low %x (will set to %x), Low rxreqmax %x, High %x (CD %p)\n",
info->source, info->low_sequence_number,
         new_lo, info->low_rxreq_max_sequence_number, info->high_sequence_number,
info->source_clientd);
     info->low_sequence_number = new_lo;
     return 0;
}
```

**JAVA API**

```
UMERcvRecInfo umerecinfocb = new UMERcvRecInfo(seqnum_offset);
rcv_attr.setRecoverySequenceNumberCallback(umerecinfocb, null);
System.out.println("Will use seqnum info with low offset " + seqnum_offset + ".");

class UMERcvRecInfo implements UMERecoverySequenceNumberCallback {
    private long _seqnum_offset = 0;

    public UMERcvRecInfo(long seqnum_offset) {
        _seqnum_offset = seqnum_offset;
    }

    public int setRecoverySequenceNumberInfo(Object cbArg,
    UMERecoverySequenceNumberCallbackInfo cbInfo)
    {
```

```
long new_low = cbInfo.lowSequenceNumber() + _seqnum_offset;
if (new_low < 0) {
System.out.println("New low sequence number would be negative.
Leaving low SQN unchanged.");
new_low = cbInfo.lowSequenceNumber();
         }
         System.out.println("SQNs Low " + cbInfo.lowSequenceNumber() + " (will set to "
                 + new_low + "), Low rxreqmax " + cbInfo.lowRxReqMaxSequenceNumber()
                 + ", High " + cbInfo.highSequenceNumber());
         try {
             cbInfo.setLowSequenceNumber(new_low);
         }
         catch (LBMEInvalException e) {
             System.err.println(e.getMessage());
        }
         return 0;
 }
```

### .NET API

```
UMERcvRecInfo umerecinfocb = new UMERcvRecInfo(seqnum_offset);
rcv_attr.setRecoverySequenceNumberCallback(umerecinfocb, null);
System.Console.Out.WriteLine("Will use seqnum info with low offset " + seqnum_offset + ".");

class UMERcvRecInfo implements UMERecoverySequenceNumberCallback {
    private long _seqnum_offset = 0;

    public UMERcvRecInfo(long seqnum_offset) {
        _seqnum_offset = seqnum_offset;
    }

    public int setRecoverySequenceNumberInfo(Object cbArg,
    UMERecoverySequenceNumberCallbackInfo cbInfo)
    {
long new_low = cbInfo.lowSequenceNumber() + _seqnum_offset;
if (new_low < 0) {
System.Console.Out.WriteLine ("New low sequence number would be negative.
Leaving low SQN unchanged.");
new_low = cbInfo.lowSequenceNumber();
         }
         System.Console.Out.WriteLine ("SQNs Low " + cbInfo.lowSequenceNumber() + "
                (will set to "
                 + new_low + "), Low rxreqmax " + cbInfo.lowRxReqMaxSequenceNumber()
                 + ", High " + cbInfo.highSequenceNumber());
         try {
             cbInfo.setLowSequenceNumber(new_low);
         }
         catch (LBMEInvalException e) {
             System.Console.Out.WriteLine (e.getMessage());
        }
         return 0;
 }
```

#### *7.1.3.5. Message Consumption*

Receivers use message consumption, defined as message deletion, to indicate that **UMP** should notify the store(s) that the application consumed the message. This notification takes the form of an acknowledgement, or ACK, to the store(s) in use as well as to the source if you configured the source for delivery confirmation (../Config/ultramessagingpersistenceoptions.html#SOURCEUMECONFIRMEDDELIVERYNOTIFICATION).

- In the C API (../API/index.html), message deletion happens by default when the receive callback returns, unless the callback uses `lbm_msg_retain()`. If the callback uses `lbm_msg_retain()` then the application has responsibility to use `lbm_msg_delete()` when it has finished processing the message.
- In the Java API (../JavaAPI/html/index.html) and .NET API (../DotNetAPI/doc/Index.html), message deletion must be triggered explicitly by the application by using the `dispose()` method of the message object. Without explicit usage of `dispose()`, **UMP** does not know when the application has finished processing the message.

##### *7.1.3.5.1. Batching Acknowledgments*

You can configure **UMP** to acknowledge message consumption to a store(s) for a series of messages independent of when the receiving application consumed the messages. This option works well if multiple threads process messages off of an event queue, which may result in messages being consumed out of order. This feature is not compatible with *Explicit Acknowledgments*.

If you set `ume_use_ack_batching` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEUSEACKBATCHING) to **1**, **UMP** does not acknowledge individual messages as the application consumes them. Instead, **UMP** checks the consumed, but unacknowledged messages at the interval configured with `ume_ack_batching_interval` (../Config/ultramessagingpersistenceoptions.html#CONTEXTUMEACKBATCHINGINTERVAL). When **UMP** discovers a contiguous series of consumed message sequence numbers (`sqn`), it sends acknowledgments to the store(s) for all the contiguous messages.

For example, assume your application consumes and acknowledges messages 1 and 2, then consumes subsequent messages in the following order: 4, 5, 7, 8, 6, 10, 3. At the next `ume_ack_batching_interval` (../Config/ultramessagingpersistenceoptions.html#CONTEXTUMEACKBATCHINGINTERVAL), **UMP** sends consumption acknowledgments to the store(s) for messages 3 - 8.

##### *7.1.3.5.2. Explicit Acknowledgments*

In addition, **UMP** supports Explicit ACKs ( `ume_explicit_ack_only` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEEXPLICITACKONLY)), which silences **UMP**'s acknowledgement behavior, allowing your application control of message consumption notification. See also `lbm_msg_ume_send_explicit_ack()` in the C API (../API/index.html) and the `LBMMessage` class method `sendExplicitAck()` in the Java API (../JavaAPI/html/index.html) and .NET API (../DotNetAPI/doc/Index.html).

The explicit ACK sending function/method automatically supplies additional ACKs for missing messages in sequence number gaps. This can be a useful efficiency feature, but note that to acknowledge each message consumption individually, you must issue their ACKs in ascending sequence-number order.

#### *7.1.3.5.3. Object-free Explicit Acknowledgments*

When using explicit ACKs in your Java (../JavaAPI/html/index.html) or .NET (../DotNetAPI/doc/Index.html) application, you can extract ACK information from messages and then send acknowledgements to the store(s) for any sequence number. You can also extract ACK information from a message when using the C API (../API/index.html) with `lbm_msg_extract_ume_ack()`.

The following source code examples show how to extract ACK information and send an explicit ACK.

**C API**

```
int rcv_handle_msg(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)
{
lbm_ume_rcv_ack_t *ack = NULL;
...

ack = lbm_msg_extract_ume_ack(msg);
lbm_ume_ack_send_explicit_ack(ack, msg->sequence_number);
lbm_ume_ack_delete(ack);
...

}
```

**JAVA API or .NET API**

```
public int onReceive(Object cbArg, LBMMessage msg)
{
UMEMessageAck ack;
...

ack = msg.extractUMEAck();
ack.sendExplicitAck(msg.sequenceNumber());
ack.dispose();

...

}
```

## 7.1.4. UMP Stores

As mentioned in *Persistent Store*, the **UMP** persistent stores, also just called stores, actually persist the source and receiver state and use RegIDs to identify sources and receivers. Each source to which a store provides persistence may have zero or more receivers. The store maintains each receiver's state along with the source's state and the messages the source has sent.

The store can be configured with its own set of options (../UME/umestored-config.html) to persist this state information on disk or simply in memory. The term *disk store* is used to signify a store that persists state to disk, and the term *memory store* is used to signify a store that persists state only in memory. A store may also be configured

not to cache the source's data, but to simply persist the source and receiver state in memory. This is called a *no-cache store*.

Unlike many persistent systems, the persistent store in **UMP** is not in the message path. In other words, a source does not send data to the store and then have the store forward it to the receivers. In **UMP** , the source sends to the receiver(s) and the store(s) in parallel. See *Normal Operation*. Thus, **UMP** can provide extremely low latency to receiving applications.

The store(s) that a source uses are part of the source's configuration settings. Sources must be configured to use specific store(s) and use one of two different types of store failover behaviors to match expected failure scenarios. See *Round-Robin Store Usage* and *Quorum/Consensus Store Usage* below for more about store failover scenarios.

Receivers, on the other hand, do not need to be configured with store information a priori. The source advertises store information as part of the normal **UM** topic resolution process. Thus the receivers will learn the store(s) to use from the source without needing to be configured themselves. Because receivers learn about the store(s) a source is using via topic resolution, the source needs to be available to receivers as long as the receivers may need them. However, the source does not have to be actively sending data to do this.

#### 7.1.4.1. Round-Robin Store Usage

Stores can be used in a Round-Robin fashion by a source during failover. A source provides **UMP** with a list of stores to use. The first is the primary, the second is the secondary, the third is the tertiary, etc. The source uses a single store at any one time. If the currently active store becomes unresponsive due to a crash or network disconnect, **UMP** tries other stores in the list one by one until it finds a responsive store.

With round-robin store usage, inactive stores do not receive data from the source. Thus, a store that becomes the active store will not have any data from the source. In this case, the source may be configured to retain messages and stream those messages to the new store using Late Join (../Config/latejoinoptions.html#SOURCELATEJOIN). Cascading failures of sources, stores and receivers may require using stores in a Quorum/Consensus fashion.

See also *Sources Using Round-Robin Store Configuration*.

#### 7.1.4.2. Quorum/Consensus Store Usage

To provide the highest degree of resiliency in the face of failures, **UMP** provides the Quorum/Consensus failover strategy which allows a source to provide **UMP** with a number of stores to be used at the same time. Multiple stores can fail and **UMP** can continue operation unhindered. Moreover, Late Join is not needed as in Round-Robin.

Quorum/Consensus, also called QC, allows a source and the associated receivers to have their persisted state maintained at several stores at the same time. Central to QC is the concept of a group of stores, which is a logical grouping of stores that are intended to signify a single entity of resilience. Within the group, individual stores may fail but for the group as a whole to be viable and provide resiliency, a quorum must be available. In **UMP** , a quorum is a simple majority. For example, in a group of five stores, three stores are required to maintain a quorum. One or two stores may fail and the group continues to provide resiliency. **UMP** requires a source to have a quorum of stores available in the group in order to send messages. A group can consist of a single store.

QC also provides the ability to use multiple groups. As long as a single group maintains quorum, then **UMP** allows a source to proceed. Groups are logical in nature and can be combined in any way imaginable, such as by store location, store type, etc. In addition, QC provides the ability to specify backup stores within groups. Backups may be used if or when a store in the group becomes unresponsive to the source. Quorum/Consensus allows a source many different failure scenarios simply not available in other persistent messaging systems.

See also *Sources Using Quorum/Consensus Store Configuration*, *Quorum/Consensus - Single Location Groups* and *Quorum/Consensus - Mixed Location Groups*.

## 7.2. Fault Recovery

Recovery from source and receiver failure is the real heart of **UMP** operation. For a source, this means continuing operation from where it stopped. For a receiver, this means essentially the same thing, but with the retransmission of missed messages. Application developers can easily leverage the information in **UMP** to make their applications recover from failure in graceful ways.

Late Join (../Design/lbm-features.html#USING-LATE-JOIN) is the mechanism of **UMP** recovery as well as an **UM** streaming feature. If Late Join is turned off on a source ( `late_join` (../Config/latejoinoptions.html#SOURCELATEJOIN)) or receiver ( `use_late_join` (../Config/latejoinoptions.html#RECEIVERUSELATEJOIN)), it also turns off **UMP** recovery. In order to control Late Join behavior, **UMP** provides a mechanism for a receiver to control the low sequence number. See *Recovery Management*.

Not all failures are recoverable. For application developers it usually pays in the long run to identify what types of errors are non-recoverable and how best to handle them when possible. Such an exercise establishes the precise boundaries of expected versus abnormal operating conditions.

> **Note: UMP** does not acknowledge messages that are lost. If the store is unable to recover a lost message, any receivers attempting to recover this message from the store will experience unrecoverable loss as well. Sources can pay attention to any gaps in stability or confirmed delivery acknowledgements as these most likely represent unrecoverable loss at the store or receivers, respectively.

This section discussed the following recovery topics.

- *Source Recovery*
- *Receiver Recovery*

### 7.2.1. Source Recovery

The following shows the basic steps of source recovery.

1. Re-register with the store.
2. Determine the highest sequence number that the store has from the source.
3. Resume sending with the next sequence number.

Because **UMP** allows you to stream messages and not wait until a message is stable at the persistent store before sending the next message, the main task of source recovery is to determine what messages the persistent store(s) have and what they don't. Therefore, when a source re-registers with a store during recovery, the store tells the source what sequence number it has as the most recent from the source. The registration event informs the application of this sequence number. See *Source Event Handler*.

In addition, a mechanism exists (`LBM_SRC_EVENT_SEQUENCE_NUMBER_INFO`) that allows the application to know the sequence number assigned to every piece of data it sends. The combination of registration and sequence number information allows an application to know exactly what a store does have and what it does not and where it should pick up sending. An application designed to stream data in this way should consider how best to maintain this information.

When QC is in use, **UMP** uses the consensus of the group(s) to determine what sequence number to use in the first message it will send. This is necessary as not all stores can be expected to be in total agreement about what was sent in a distributed system. The application can configure the source with the `ume_consensus_sequence_number_behavior` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMECONSENSUSSEQUENCENUMBERBEHAVIOR ) to use the lowest sequence number of the latest group of sequence numbers seen from any store, the highest, or the majority. In most cases, the majority, which is the default, makes the most sense as the consensus. The lowest is a very conservative setting. And the highest is somewhat optimistic. Your application has the flexibility to handle this in any way needed.

If streaming is not what an application desires due to complexity, then it is very simple to use the **UMP** events (*UMP and UMQ Events*) delivered to the application to mimic the behavior of restricting a source to having only one unstable message at a time.

### 7.2.2. Receiver Recovery

The following shows the basic steps of receiver recovery.

1. Re-register with the store.
2. Determine the low sequence number.
3. Request retransmission of messages starting with the low sequence number.

**UMP** provides extensive options for controlling how receivers handle recovery. By default, receivers want to restart after the last piece of data that was consumed prior to failure or graceful suspension. Since **UMP** persists receiver state at the store, receivers request this state from the store as part of re-registration and recovery.

The actual sequence number that a receiver uses as the first topic level message to resume reception with is called the "low sequence number". **UMP** provides a means of modifying this sequence number if desired. An application can decide to use the sequence number as is, to use an even older sequence number, to use a more recent sequence number, or to simply use the most recent sequence number from the source. See *Recovery Management* and *Setting Callback Function to Set Recovery Sequence Number*. This allows receivers great flexibility on a per source basis when recovering. New receivers, receivers with no pre-existing registration, also have the same flexibility in determining the sequence number to begin data reception.

Like sources, when QC is in use, **UMP** uses the consensus of the group(s) to determine the low sequence number. And as with sources, this is necessary as not all stores can be expected to be in total agreement about what was acknowledged. The application can configure the receiver with `ume_consensus_sequence_number_behavior` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMECONSENSUSSEQUENCENUMBERBEHAVIOR) to use the lowest sequence number of the latest group of sequence numbers seen from any store, the highest, or the majority. In most cases, the majority, which is the default, makes the most sense as the consensus. The lowest is a very conservative setting. And the highest is somewhat optimistic. In addition, this sequence number may be modified by the application after the consensus is determined.

For QC, **UMP** load balances receiver retransmission requests among the available stores. In addition, if requests are unanswered, retransmissions of the actual requests will use different stores. This means that as long as a single store has a message, then it is possible for that message to be retransmitted to a requesting receiver.

> **Note:** Receivers need to consider if the use of arrival order delivery is appropriate. See `ordered_delivery` (../Config/majoroptions.html#RECEIVERORDEREDDELIVERY). **UMP** stores save the highest sequence number acknowledged by a receiver. When receivers using arrival order delivery receive - and thereby acknowledge - messages out of order, recovery problems may arise because stores will not have earlier messages not acknowledged by the receiver.

# 8. Enabling Queuing

This section describes how to add queuing to a minimum **UM** source and receiver with the following.

- *Starting Configuration*
- *Adding a Queue to a Source*
- *Sending to the Queue*
- *Receiving from the Queue*

**UMQ** Prerequisite: You should understand basic **Ultra Messaging** concepts such as Sources and Receivers along with the basic methods for configuring them.

The following table lists all source files used in this section. You can also find links to them in the appropriate task. The files can also be found in the `/doc/UME` directory.

| Object | Filename |
|---|---|
| Source Application | q-example-src.c |
| Receiver Application | q-example-rcv.c |
| **UMQ** Configuration File | q-example-config.xml |

## 8.1. Starting Configuration

We begin with the minimal source and receiver used by the UM QuickStart Guide (../QuickStart/index.html). To more easily demonstrate the **UMQ** features we are interested in, we have modified the QuickStart source and receiver in the following ways.

- Modified the source to send 20 messages with a one second pause between each message.
- Modified the receiver to anticipate 20 messages instead of just one.
- Assigned the topic, **UME Queue Example**, to both the source and receiver.
- Modified the receiver to not exit on unexpected receiver events.

The last change allows us to better demonstrate basic operation and evolve our receiver slowly without having to anticipate all the options that **UMQ** provides up front.

> **Note:** Be sure to build q-example-src.c and q-example-rcv.c. Instructions for building them are at the beginning of the source files.

## 8.2. Adding a Queue to a Source

To enable queuing, a source needs to be configured to use a queue by setting the `umq_queue_name` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQQUEUENAME) for the source. The following q-example-src.c code sets the source's queue to the `queue_name` established in the **UMQ** configuration file (`q-example-config.xml` ).

```
err = lbm_src_topic_attr_str_setopt(&attr, "umq_queue_name", queue_name);
```

Examining q-example-config.xml reveals that `queue_name` is set to **Queue Training** and listens on port 14567 for registrations.

Let's instantiate the queue with the following command.

```
$ umestored q-example-config.xml
```

## 8.3. Sending to the Queue

Further examination of q-example-config.xml shows that the **Queue Training** is configured for *Serial Queue Dissemination (SQD)*. In this model, sources send messages to topics as usual, but receivers interested in the topic need not even be started because topic messages go to the Queue first.

Therefore we can take the next step and run the source with the following command.

```
$ q-example-src
```

The sample output below shows that the source sends 20 messages on the **UME Queue Example** topic.

```
tmont@agentmbp:/Volumes/source$ ./q-example-src
INFO: Host has multiple multicast-capable interfaces. Default multicast interface: [en1][192.168.0.4
INFO: Source "UME Queue Example" Late Join not set, but UME store or UMQ queue specified. Setting La
Sent Message 01
Sent Message 02
Sent Message 03
Sent Message 04
Sent Message 05
Sent Message 06
Sent Message 07
Sent Message 08
Sent Message 09
Sent Message 10
Sent Message 11
```

```
Sent Message 12
Sent Message 13
Sent Message 14
Sent Message 15
Sent Message 16
Sent Message 17
Sent Message 18
Sent Message 19
Sent Message 20
```

## 8.4. Receiving from the Queue

At this point, only the Queue is running. The receiving application can start up at anytime and receive any messages from the Queue on the topic it subscribes to.

```
tmont@agentmbp:/Volumes/source$ ./q-example-rcv
INFO: Host has multiple multicast-capable interfaces. Default multicast interface: [en1][192.168.0.4
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 01'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 02'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 03'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 04'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 05'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 06'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 07'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 08'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 09'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 10'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 11'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 12'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 13'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 14'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 15'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 16'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 17'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 18'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 19'
Received 20 bytes on topic UME Queue Example: 'UME Queue Message 20'
```

# 9. Designing Queuing Applications

**UMQ** applications are much like **UMP** persistence applications, but with even fewer recovery concerns.

- First, **UMQ** receivers typically have no concept of recovery. In queuing semantics, the individual receiver has no requirement to pick up where it left off. By default, the Queue assigns another receiver if the original receiver goes away. However, you can also configure a queue to avoid reassignment or to never reassign.

- Secondly, sources either submit messages to queues or they don't. Sources that fail before a queue can acknowledge the messages as stable should simply resend the messages once recovered. In this regard, the same recovery sequence for sources at a high level can be applied to **UMQ** sources. Thus recovery from failure for applications is fairly straightforward.

## 9.1. Queue Registration IDs

Each context belonging to a **UMQ** source or receiver application registers with a queue. This registration uses a Registration ID, which is not used in the same manner as with **UMP** persistent stores (which register sources and receivers). **UMQ** Registration IDs identify individual contexts and not individual source and receiver objects. Also, **UMQ** Registration IDs can (and should) vary per invocation. Applications can set the use of specific Registration IDs with specific queues if they desire. But we recommend you let **UMQ** generate these Registration IDs or use *Queue Session IDs* to generate/manage them.

**UMQ** directs the registration of contexts with queues internally. The following is a high level description of the registration sequence.

- Context creates a source object with `umq_queue_name` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQQUEUENAME) set to the desired Queue.
- The context resolves the Queue using Queue Information Records (QIR) and Queue Query Records (QQR).
- The context generates a Registration ID randomly or uses one provided by your application from the `lbm_umq_queue_entry_t` for the context.
- **UMQ** registers the context with the Queue by sending the Registration ID to be used to the Queue.

## 9.2. Queue Session IDs

Like **UMP**, you can use Session IDs to manage context registration IDs and receiver assignment IDs. The Session ID is a 64-bit value that that identifies a receiving context and its set of receivers for a particular topic. A single Session ID allows queues to correctly identify all receivers for a particular application.

With Session IDs, a receiver that fails can return with its original assignment ID and continue to receive queued messages, and receive them in the correct order. In this scenario, with option `message-reassignment-timeout` (see *Options for a Topic's ume-attributes Element*) set to 0 (never reassign), the queue continues to send queued messages to the same designated receiver in the designated order, even in the event of receiver failure and recovery.

To set the queue Session ID, set option `umq_session_id` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQSESSIONID) to a unique value. Do not replicate this value elsewhere, even for sending applications.

## 9.3. Message IDs

**UMQ** assigns each message a unique Message ID (`lbm_umq_msgid_t`), which is composed of the sending application's context Registration ID and a unique, incrementing stamp. These Message IDs must be unique for a given queue, however, the application can use them for a variety of processing purposes.

## 9.4. Message Lifetimes and Reassignment

Because receivers may be assigned messages and have failures before they can consume a message (or fail while consuming a message), queues use the following parameters to control how long a message should take to be consumed.

- total lifetime
- reassignment timeout
- maximum reassignments

If the assigned receiver does not consume a message after the reassignment timeout expires, the queue reassigns the message to another receiver provided the total lifetime has not expired.

A queue can reassign an unconsumed message repeatedly until either it reaches the maximum reassignments or the total lifetime expires. The queue marks reassigned messages as having been re-assigned. Receivers may use this re-assignment flag as a hint that they may want to treat the message differently.

The lifetime begins when the queue first assigns the message. When the total lifetime expires, the queue either discards the message from the queue permanently or sends it to the *Dead Letter Queue*, if configured.

You can set a message lifetime in the following ways.

- **For a Source** - To set a message lifetime default for a particular source, set `umq_msg_total_lifetime` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQMSGTOTALLIFETIME) to the number of milliseconds after which the message should not be assigned. Set `message-reassignment-timeout` and `message-max-reassignments` in *Options for an Application Set's ume-attributes Element*.
- **For a Source Sending to ULB Receivers** - Set the message lifetime default with `umq_ulb_application_set_message_lifetime` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBAPPLICATIONSETMESSAGELIFETIME). Set the reassignment timeout with `umq_ulb_application_set_message_reassignment_timeout` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBAPPLICATIONSETMESSAGEREASSIGNMENTTIMEOUT). Set the maximum reassignments with `umq_ulb_application_set_message_max_reassignments` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQULBAPPLICATIONSETMESSAGEMAXREASSIGNMENTS).
- **For a Queue Topic** - Set the `umestored` Queue Topic attribute, `message-total-lifetime`, to the number of milliseconds after which the message should not be assigned. Set `message-reassignment-timeout` and `message-max-reassignments` in *Options for a Queue Topic's ume-attributes Element*.
- **For a particular message send** - Use an extended send call, `lbm_src_send_ex`, that includes a pointer to `lbm_umq_msg_total_lifetime_info_t` (../API/structlbm__umq__msg__total__lifetime__info__t__stct.html) in `lbm_src_send_ex_info_t`. Set the `umq_msg_total_lifetime` member to a `lbm_umq_msg_total_lifetime_info_t` structure and set the `lbm_umq_msg_total_lifetime_info_t.umq_msg_total_lifetime` to override the total lifetime configured via `umq_msg_total_lifetime` (../Config/ultramessagingqueuingoptions.html#SOURCEUMQMSGTOTALLIFETIME).
- **For MIM** - To set a message lifetime default for Multicast Immediate Messages (MIM), set the message lifetime option for the context, `umq_msg_total_lifetime` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQMSGTOTALLIFETIME), to the number of milliseconds after which the message should not be assigned.

A receiving application can pre-empt reassignment configurations by using `lbm_msg_umq_reassign()` for either **UMQ** or ULB receivers. This function takes an `lbm_msg_t` and flags for arguments. With no flags set, the queue reassigns the message. With the LBM_MSG_UMQ_REASSIGN_FLAG_DISCARD flag set, the queue discards the message. The corresponding Java and .NET methods are `LBMMessage.reassign()`.

# 10. Fault Tolerance

This section discusses the following.

- *Configuring for Persistence and Recovery*
- *Proxy Sources*
- *Queue Redundancy*
- *Queue Failover*

## 10.1. Configuring for Persistence and Recovery

Deployment decisions play a huge role in the success of any persistent system. Configuration in **UMP** has a number of options that aid in performance, fault recovery, and overall system stability. It is not possible, or at least not wise, to totally divorce configuration from application development for high performance systems. This is true not only for persistent systems, but for practically all distributed systems. When designing systems, deployment considerations need to be taken into account. This section discusses the following deployment considerations.

- *Source Considerations*
- *Receiver Considerations*
- *Store Configuration Considerations*
- *UMP Configuration Examples*

### 10.1.1. Source Considerations

Performance of sources is heavily impacted by:

- the release policy that the source uses
- streaming methods of the source
- the throughput and latency requirements of the data

Source release settings have a direct impact on memory usage. As messages are retained, they consume memory. You reclaim memory when you release messages. Message stability, delivery confirmation and retention size all interact to create your release policies. **UMP** provides a hard limit on the memory usage. When exceeded, a Forced Reclamation event is delivered. Thus applications that anticipate forced reclamations can handle them appropriately. See also *Source Message Retention and Release*.

How the source streams data has a direct impact on latency and throughput. One streaming method sets a maximum, outstanding count of messages. Once reached, the source does not send any more until message stability notifications come in to reduce the number of outstanding messages. The `umesrc` (../example/umesrc.c) example program uses this mechanism to limit the speed of a source to something a store can handle comfortably. This also provides a maximum bound on recovery that can simplify handling of streaming source recovery.

The throughput and latency requirements of the data are normal **UM** concerns. See **Ultra Messaging Concepts** (../Design/index.html).

## 10.1.2. Receiver Considerations

In addition to the following, receiver performance shares the same considerations as receivers during normal operation.

### *10.1.2.1. Acknowledgement Generation*

Receivers in a persistence implementation of **UMP** send an a message consumption acknowledgement to stores and the message source. Some applications may want to control this acknowledgement explicitly themselves. In this case, `ume_explicit_ack_only` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEEXPLICITACKONLY) can be used.

### *10.1.2.2. Controlling Retransmission*

Receivers in **UMP** during fault recovery are another matter entirely. Receivers send retransmission requests and receive and process retransmissions. Control over this process is crucial when handling very long recoveries, such as hundreds of thousands or millions of messages. A receiver only sends a certain number of retransmission requests at a time.

This means that a receiver will not, unless configured to with `retransmit_request_outstanding_maximum` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTOUTSTANDINGMAXIMUM), request everything at once. The value of the low sequence number (*Receiver Recovery*) has a direct impact on how many requests need to be handled. A receiving application can decide to only handle the last X number of messages instead of recovering them all using the option, `retransmit_request_maximum` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTMAXIMUM). The timeout used between requests, if the retransmission does not arrive, is totally controllable with `retransmit_request_interval` (../Config/latejoinoptions.html#RECEIVERRETRANSMITREQUESTINTERVAL). And the total time given to recover all messages is also controllable.

### *10.1.2.3. Recovery Process*

Theoretically, receivers can handle up to roughly 2 billion messages during recovery. This limit is implied from the sequence number arithmetic and not from any other limitation. For recovery, the crucial limiting factor is how a receiver processes and handles retransmissions which come in as fast as **UMP** can request them and a store can retransmit them. This is perhaps much faster than an application can handle them. In this case, it is crucial to realize that as recovery progresses, the source may still be transmitting new data. This data will be buffered until recovery is complete and then handed to the application. It is prudent to understand application processing load when planning on how much recovery is going to be needed and how it may need to be configured within **UMP** .

## 10.1.3. Store Configuration Considerations

**UMP** stores have numerous configuration options. See *Configuration Reference for Umestored*. This section presents issues relating to these options.

### *10.1.3.1. Configuring Store Usage per Source*

A store handles persisted state on a per topic per source basis. Based on the load of topics and sources, it may be prudent to spread the topic space, or just source space, across stores as a way to handle large loads. As configuration of store usage is per source, this is extremely easy to do. It is easy to spread CPU load via multi-threading as well as hard disk usage across stores. A single store process can have a set of virtual stores within it, each with their own thread.

### *10.1.3.2. Disk vs. Memory*

As mentioned previously in *UMP Stores*, stores can be memory based or disk based. Disk stores also have the ability to spread hard disk usage across multiple physical disks by using multiple virtual stores within a single store process. This gives great flexibility on a per source basis for spreading data reception and persistent data load.

**UMP** stores provide settings for controlling memory usage and for caching messages for retransmission in memory as well as on disk. See *Options for a Topic's ume-attributes Element*. All messages in a store, whether in memory or on disk, have some small memory state. This is roughly about 72 bytes per message. For very large caches of messages, this can become non-trivial in size.

### *10.1.3.3. Activity Timeouts*

**UMP** stores are NOT archives and are not designed for archival. Stores persist source and receiver state with the aim of providing fault recovery. Central to this is the concept that a source or receiver has an activity timeout attached to it. Once a source or receiver suspends operation or has a failure, it has a set time before the store will forget about it. This activity timeout needs to be long enough to handle the recovery demands of sources and receivers. However, it can not and should not be infinite. Each source takes up memory and disk space, therefore an appropriate timeout should be chosen that meets the requirements of recovery, but is not excessively long so that the limited resources of the store are exhausted.

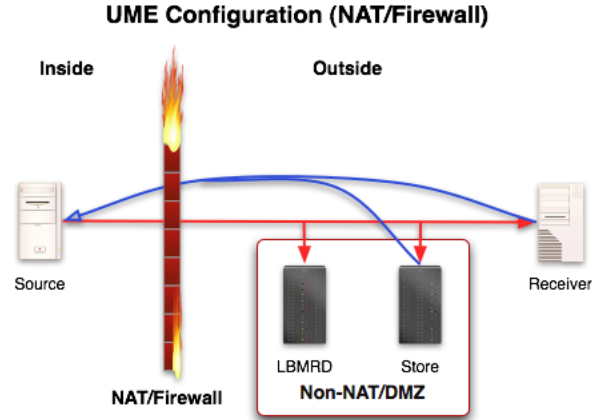
## 10.1.4. UMP Configuration Examples

The following example configurations are offered to illustrate some of the many options available to configuring **UMP** .

### *10.1.4.1. **UMP** Configuration with NAT/Firewall*

Although the diagram, *Normal Operation*, demonstrates the typical message interaction in **UMP** , sources, receivers, and stores may be arranged in almost limitless configurations. Some configurations make more sense than others for certain situations. One of those situations involves a Network Address Translation configuration (NAT) and/or Firewall. In such configurations, the source is the key element behind the NAT or Firewall. Although not the only viable NAT/Firewall configuration for **UMP** , the figure below demonstrates one approach to such an arrangement.

**Figure 17. UMP Configuration with NAT/Firewall**




The `lbmrd` (29west Resolution Daemon) is an optional piece, but used in most situations where a NAT or Firewall is involved. It provides unicast support for topic resolution. The `lbmrd` and the store are placed on the outside (or at least are non-NATed or on a DMZ). Important characteristics of this configuration are:

- The LBMRD acts as a proxy for the topic resolution information.
- The store is accessible by the source and receiver directly.

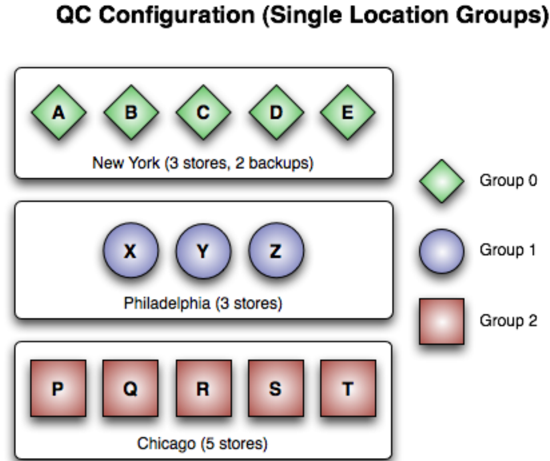
In this situation, receivers and stores unicast control information back to the source, therefore the NAT or Firewall router needs to port forward information back to the source.

#### *10.1.4.2. Quorum/Consensus - Single Location Groups*

Quorum/Consensus provides a huge set of options for store arrangements in **UMP** . Between backups and groups, the number of viable approaches is practically limitless. Below are two approaches using single location groups and multiple location groups.

In short, as long as one of the groups in the figure maintains quorum, then the source can continue. See *Sources Using Quorum/Consensus Store Configuration* to view a **UM** configuration file for this example.

**Figure 18. Quorum/Consensus - Single Location Groups**



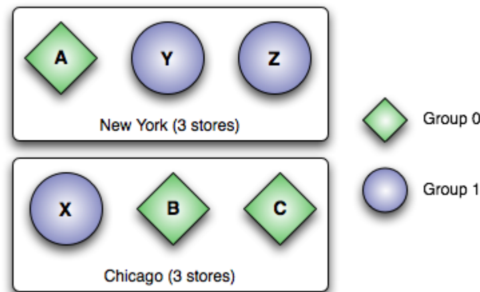

The above figure shows three groups arranged on a location basis. Each group is a single location. Just SOME possible failure scenarios are:

- Failure of any 3 stores in Group 0
- Failure of any 1 store in Group 1
- Failure of any 2 stores in Group 2
- Failure of all stores in Group 0 and 1
- Failure of all stores in Group 1 and 2
- Failure of all stores in Group 0 and 2

#### 10.1.4.3. Quorum/Consensus - Mixed Location Groups

Groups of stores can be configured across locations. Such an arrangement would ensure continued operation in the event of a site-wide failure at any location.

**Figure 19. Quorum/Consensus - Mixed Location Groups**




The figure above shows two groups arranged in a mixed location manner. Essentially, one location can totally fail and a source can continue sending because the other location has a group with a quorum. See below for an **UM** configuration file for this example.

```
source ume_store 10.16.3.77:10313:101000:0
source ume_store 10.16.3.78:11313:110000:1
source ume_store 10.16.3.79:12313:120000:1
source ume_store 192.168.0.44:15313:150000:1
source ume_store 192.168.0.45:16313:160000:0
source ume_store 192.168.0.46:17313:170000:0

source ume_message_stability_notification 1
source ume_store_behavior qc

source ume_store_group 0:3
source ume_store_group 1:3

source ume_retention_intragroup_stability_behavior quorum
source ume_retention_intergroup_stability_behavior any
```

## 10.2. Proxy Sources

The Proxy Source capability allows you to configure stores to automatically continue sending the source's topic advertisements which contain store information used by new receivers. Without the store RegID, address and TCP port contained in the source's Topic Information Records (TIR), new receivers cannot register with the store or request retransmissions. After the source returns, the store automatically stops acting as a proxy source.

Some other features of Proxy Sources include:

- Requires a Quorum/Consensus store configuration.

- Normal store failover operation also initiates a new proxy source.
- A store can be running more than one proxy source if more than one source has failed.
- A store can be running multiple proxy sources for the same topic.

## 10.2.1. How Proxy Sources Operate

The following sequence illustrates the life of a proxy source.

1. A source configured for Proxy Source sends to receivers and a group of Quorum/Consensus stores.
2. The source fails.
3. The source's `ume_activity_timeout` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEACTIVITYTIMEOUT) or the store's `source-activity-timeout` expires.
4. The Quorum/Consensus stores elect a single store to run the proxy source.
5. The elected store creates a proxy source and sends topic advertisements.
6. The failed source reappears.
7. The store deletes the proxy source and the original source resumes activity.

If the store running the proxy source fails, the other stores in the Quorum/Consensus group detect a source failure again and elect a new store to initiate a proxy source.

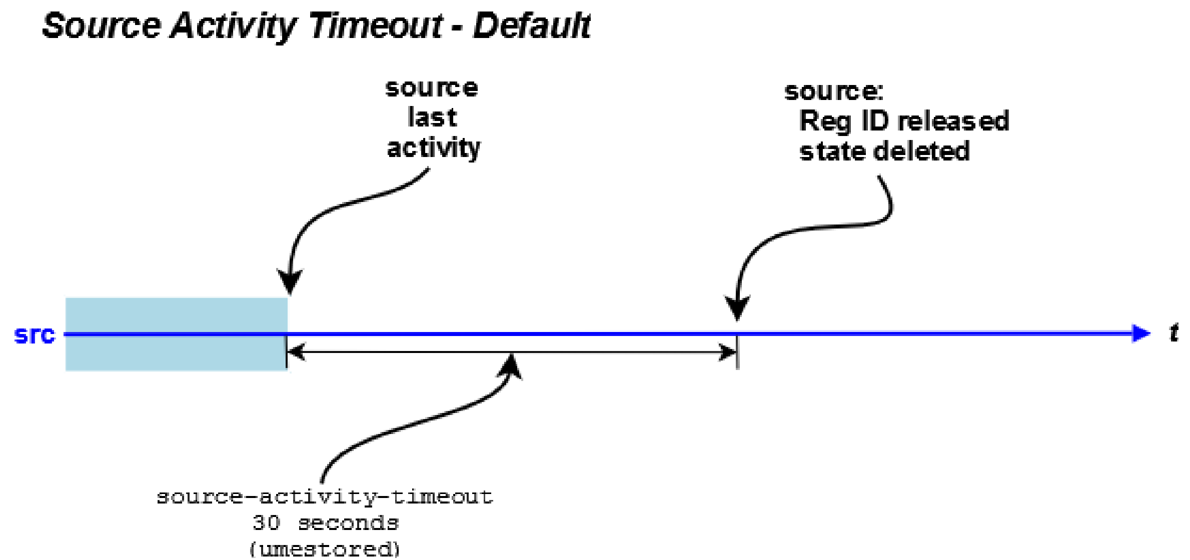
If a loss of quorum occurs, the proxy source can continue to send advertisements, but cannot send messages until a quorum is re-established.

## 10.2.2. Activity Timeout and State Lifetime Options

**UMP** provides activity and state lifetime timers for sources and receivers that operate in conjunction with the proxy source option or independently. This section explains how these timers work together and how they work with proxy sources.

The `ume_activity_timeout` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEACTIVITYTIMEOUT) options determine how long a source or receiver must be inactive before a store allows another source or receiver to register using that RegID. This prevents a second source or receiver from *stealing* a RegID from an existing source or receiver. An activity timeout can be configured for the source/receiver with the **UM** Configuration Option cited above or with a topic's `ume-attribute` configured in the `umestored` XML configuration file. The following diagram illustrates the default activity timeout behavior, which uses `source-state-lifetime` in the `umestored` XML configuration file. (See *Options for a Topic's ume-attributes Element*.)

**Figure 20. Source Activity Timeout Default**

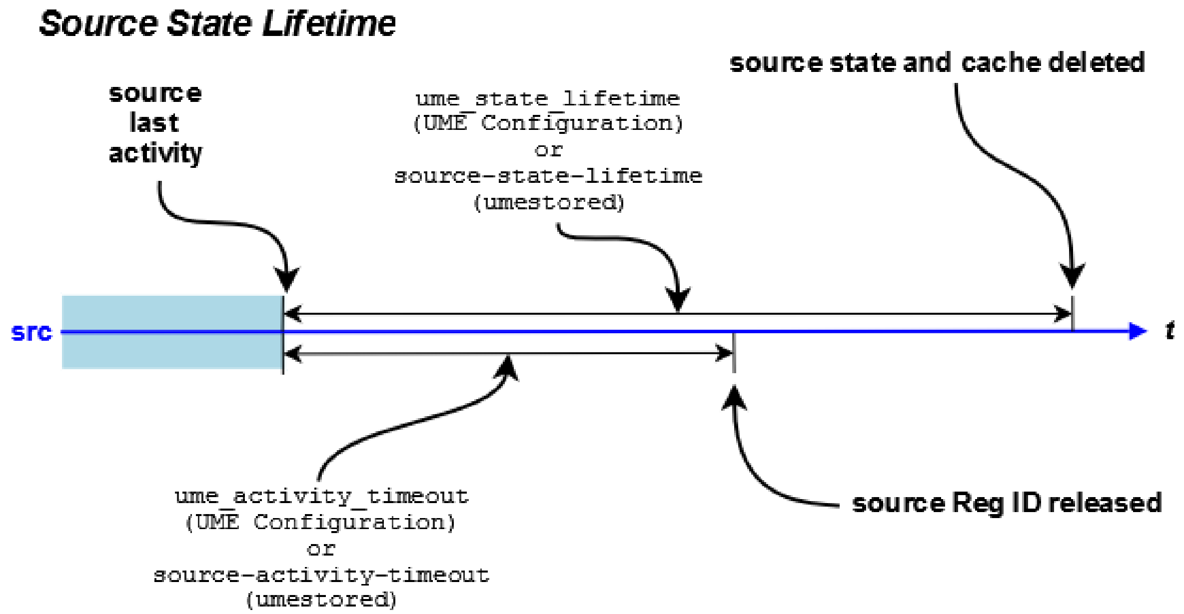



In addition to the activity timeout, you can also configure sources and receivers with a state lifetime timer using the following options.

- `(source) ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTATELIFETIME)
- `(receiver) ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMESTATELIFETIME)
- The topic's ume-attributes options, `source-state-lifetime` and `receiver-state-lifetime`. See *Options for a Topic's ume-attributes Element*.

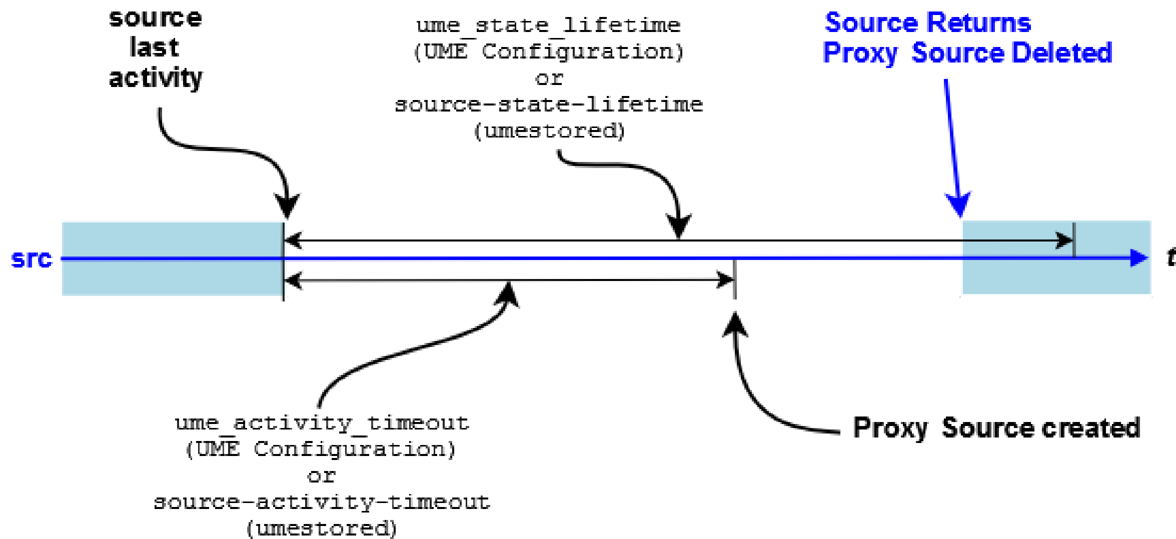
The `ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTATELIFETIME), when used in conjunction with the `ume_activity_timeout` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEACTIVITYTIMEOUT) options, determines at what point **UMP** removes the source or receiver state. **UMP** does not check the `ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTATELIFETIME) until `ume_activity_timeout` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEACTIVITYTIMEOUT) expires. The following diagram illustrates this behavior.

**Figure 21. Source or Receiver State Lifetime**




If you have enabled the Proxy Source option, the Activity Timeout triggers the creation of the proxy source. The following diagram illustrates this behavior.

**Figure 22. Source Activity and State Timers with the Proxy Source Option**




## 10.2.3. Enabling the Proxy Source Option

You must configure both the source and the stores to enable the Proxy Source option.

- Configure the source in a **UM** Configuration File with the source configuration option, `ume_proxy_source` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEPROXYSOURCE).
- Configure the stores in the `umestored` XML configuration file with the Store Element Option, `allow-proxy-source`. See *Options for a Store's ume-attributes Element* for more information.

**Note:** Proxy sources operate with Session IDs as well as Reg IDs. See *Managing RegIDs with Session IDs*

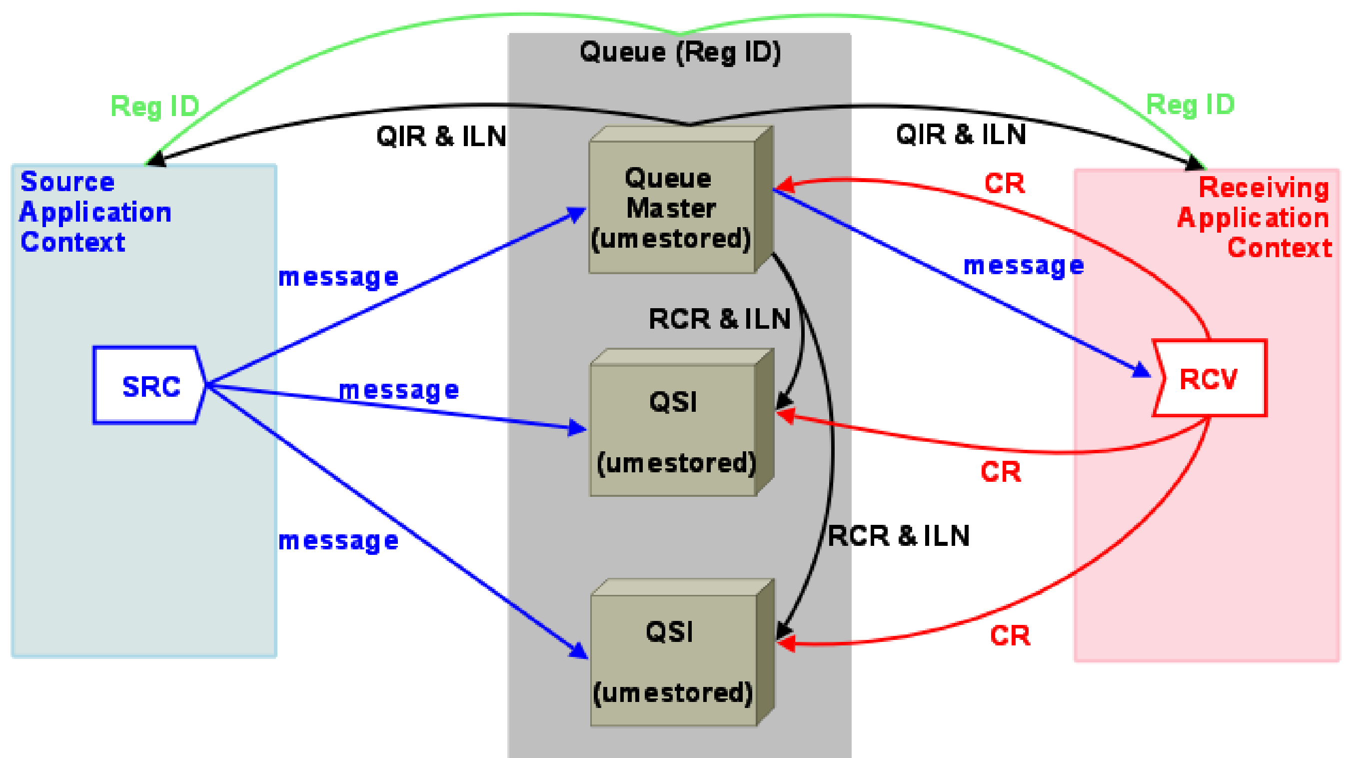
## 10.3. Queue Redundancy

Queues can use the same Quorum/Consensus configuration as **UMQ** stores.

- Sources submit queue messages to each queue instance.
- You configure queue instances into groups using the *Queue Groups Element* of the queue's `umestored` XML configuration file. You can configure all queue instances for a queue in a single `umestored` XML configuration file or a separate file for each instance.
- A message is considered stable once it has satisfied the stability requirements you configure with `umq_retention_*_stability_behavior` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQRETENTIONINTERGROUPSTABILITYBEHAVIOR).
- Receivers send Consumption Reports to all queue instances so they all are aware of message reception status.

Figure 23 shows how multiple instances of a Queue can be configured and some of the communication between contexts, sources, receivers and Queue instances.

**Figure 23. Sample Configuration of Redundant Queues for Failover**



- **RegID** - Application contexts register with the Queue using a RegID in order to send messages to the Queue or receive messages from the Queue. See *Queue Registration IDs* and `umq_queue_registration_id` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQQUEUEREGISTRATIONID).
- **Queue Master** - A Queue Instance that has been elected the master Queue. Only the master Queue sends messages to receivers. The master sends Instance List Notifications (ILN) periodically to QSIs and registered contexts.
- **Queue Slave Instance (QSI)** - Slave Queue Instances initiated for Queue Failover. Should the master Queue fail, any QSI can assume master Queue activities following election. Each QSI and the master Queue require a separate **UM** Configuration File and `umestored` configuration.
- **QIR (Queue Information Record)** - Queue advertisement sent by the master Queue to registered contexts which enable sources and receivers to resolve queue names to lists of queue instances (ILN). QIRs contain each queue instances' IP address, port, index, and group index as well as flags indicating if the instance is the Queue Election Master, current master or Post Election Master. The QIR also contains versioning information for instances partitioned due to previous failures.
- **ILN (Instance List Notification)** - List of active instances in the Queue used by instances to manage themselves. For each instance the list contains the instance's IP address, port, index, group index and flags indicating if the instance is the Queue Election Master, current master or Post Election Master.
- **CR (Consumption Report)** - Indication sent to each Queue Instance that a message has been consumed.

### 10.3.1. Queue Master Election Process

Queue instances use an internal election process to determine a master queue responsible for making assignments and handling any dissemination requirements. The master is also responsible for tracking queue instance liveness and handling queue resolution duties. Those queue instances that are not the master (slaves) simply act as passive observers of queue activities. Slaves may fail and come online without seriously impacting operations. When a master fails, though, an election occurs. Once the election process establishes a new master, queue operation can proceed.

A queue instance is elected master based on the three values shown below and presented in order of importance.

1. The QSI's `queue-management-election-bias` configured in it's `umestored` XML configuration file.
2. The QSI's age computed from the number of messages received and submitted by the QSI. **UMQ** uses the QSI age if all QSI bias values are equal.
3. An internal QSI index. **UMQ** uses the QSI index if all QSI bias and age values are equal.

**UMQ**'s default behavior assigns the same election bias to every QSI, which often results in the "oldest", active QSI being elected the master queue. If you wish finer control of the election process, you can configure each Queue Instance with an election bias. You could assign the higher election bias values to the Queue Instances you know are running on the more powerful machines or those with the lowest latency. See *Queue Management Options for a Queue's ume-attributes Element* for more information.

The following summarizes the Queue Master Election Process.

1. A Queue Slave Instance (QSI) detects the loss of the master if the `queue_management_master_activity_timeout` expires without any Instance Lists having been sent during the timeout period.
2. The QSI that detected the loss of the master names itself Queue Election Master (QEM).

3. The QEM sends an Election Call to all QSIs, which also identifies itself as the new QEM.
4. All QSIs reply to the Election Call with their "instance vote" which contains their own election bias and "age".
5. The QEM selects the QSI with the highest election bias as the master. If two or more QSIs have the highest bias, the QEM selects the "oldest" of the QSIs with the highest bias. The QSI with the most messages received and submitted is considered the oldest. A third tie breaker is an internal QSI index.
6. The QEM sends out another ILN naming the elected QSI as the Post Election Master (PEM).
7. QSIs confirm receipt of the ILN.
8. QEM sends a Resume Operation message to the PEM.
9. The PEM resumes operation of the Queue (assigning messages to receivers, managing dissemination requirements, tracking QSI health, handling queue resolution traffic) and sends a Instance List Notification flagging itself as the current master.

## 10.4. Queue Failover

The following sections discuss various queue failover scenarios.

### 10.4.1. Failover from Loss of Quorum

If the loss of a QSI results in the loss of quorum, the master Queue stops advertising (QIR). As a result, sources lose their registration and subsequent messages sent by the sources return an LBM_EUMENOREG event. (If a source is connected to both a store and a queue, subsequent message sends return an LBM_ENO_QUEUE_REG event.) When quorum is regained by the recovery of the lost QSI or the addition of a new QSI, the master Queue advertises again. Sending and receiving **UM** contexts can then resolve the Queue again and re-register.

### 10.4.2. Failover from Loss of Master

If the master Queue fails, the following two events occur.

- Sources lose their registration and subsequent messages sent by the sources return an LBM_EUMENOREG event. (If a source is connected to both a store and a queue, subsequent message sends return an LBM_ENO_QUEUE_REG event.)
- The first QSI to detect the loss of the master calls an election. See *Queue Master Election Process*. After wards, the new master Queue starts advertising, allowing sending and receiving **UM** contexts to resolve the Queue and re-register.

If, due to a series of failures, a QSI notices that it has stored messages that the master queue never saw, it attempts to resubmit them to the master queue. The master queue either accepts these when it determines that it never saw them, or it rejects the resubmission attempt. If the master queue accepts the resubmit, it marks the message as a resubmission when it assigns the message to a receiver, informing the receiver that it was resubmitted from a QSI.

### 10.4.3. Other Scenarios

If a receiver fails unexpectedly, the queue does not become aware of this until `receiver-activity-timeout` expires. In the mean time, the queue continues to assign messages to the receiver until the receiver's portion size is met. When the `message-reassignment-timeout` expires, the queue reassigns the message to a different receiver and sets the message's reassigned flag to inform the receiving application that the message may have been seen by a different receiver.

Receivers can call `lbm_rcv_umq_deregister()` or `lbm_wildcard_rcv_umq_deregister()` to gracefully deregister from the queue. The queue does not assign any new messages to it.

### 10.4.4. Failover from Loss of Slave (QSI)

Assuming the master queue is running and assuming quorum has been maintained, QSIs coming and going have little to no impact on queue operation. QSIs are largely passive bystanders. As QSIs come and go from the queue, the master queue notifies the **UM** contexts registered with the queue via instance list notifications (ILN). These notifications inform the contexts which QSI was added or removed.

# 11. Man Pages

# umestored

## Name

`umestored` — Persistent Store and Queue Daemon

## Synopsis

`umestored` [`-d`] [`--dump-dtd`] [`-f`] [`--detach`] [`-h`] [`--help`] [`-v`] [`--validate`] *`configfile`*

## Description

Persistent Store and Queue services are provided by `umestored`. A store configuration file is required.

The DTD used to validate a configuration file will be dumped to standard output with the `-d` or `--dump-dtd`. After dumping the DTD, `umestored` exits instead of providing persistence services as usual.

The configuration file will be validated against the DTD if either the `-v` or `--validate` options are given. After attempting validation, `umestored` exits instead of providing persistence services as usual. The exit status will be 0 for a configuration file validated by the DTD and non-zero otherwise.

`Umestored` normally remains attached to the controlling terminal and runs until interrupted. If the `-f` or `--detach` options are given, `umestored` instead forks, detaches the child from the controlling terminal, and the parent exits immediately.

Command line help is available with `-h`.

## Usage Notes

When shutting down the **UM** Persistent Store or Queue daemon, use a SIGINT to trigger a clean shutdown, which attempts to cleanly finish outstanding IO requests before shutting down. Two successive SIGINTs force an immediate shutdown.

## Exit Status

The exit status from `umestored` is 0 for success and some non-zero value for failure.

# umestoreds

## Name

`umestoreds` — Persistent Store and Queue Windows Service

## Synopsis

`umestoreds` [`-d`] [`--dump-dtd`] [`-h`] [`--help`] [`-s action`] [`--service=action`] [`-v`] [`--validate`] [`configfile`]

## Description

Persistent Store and Queue services are provided by the `umestoreds` Windows Service. A store configuration file is optional. If not present, the Registry will be consulted.

The DTD used to validate a configuration file will be dumped to standard output with the `-d` or `--dump-dtd`. After dumping the DTD, `umestoreds` exits instead of providing persistence services as usual.

The configuration file will be validated against the DTD if either the `-v` or `--validate` options are given. After attempting validation, `umestoreds` exits instead of providing persistence services as usual. The exit status will be 0 for a configuration file validated by the DTD and non-zero otherwise.

The `-s install` or `--service=install` options will install the service using the given configuration file. Once installed, `umestoreds` exits. Once installed, the service may be stopped or started via the Windows Service Control Panel.

The `-s remove` or `--service=remove` options will remove the service. Once removed, `umestoreds` exits.

The `-s config` or `--service=config` options will update the configuration file used with the service to be the given configuration file. Once updated, `umestoreds` exits.

Command line help is available with `-h`.

## Usage Notes

When installing the **UM** Persistent Store or Queue as a Microsoft® Windows® service, use only local disk devices and fully qualified path names for all filenames. This is because Windows services run by default under a Local System account, which has reduced privileges and is not allowed access to network devices.

Stopping the **UM** Persistent Store or Queue service triggers a clean shutdown, which attempts to cleanly finish outstanding IO requests before shutting down.

## Exit Status

The exit status from `umestored` is 0 for success and some non-zero value for failure.

# umqsltool

## Name

`umqsltool` — UMQ SINC Log Tool

## Synopsis

`umqsltool` [`options`] *`sinclogfile1`* [*`sinclogfile2`*]

## Description

This command provides SINC log file tools that let you dump to text, compare two files, or trim events to reduce file size (all without modifying the original log files).

The `-t` *toolname* or `--tool=`*toolname* invokes the desired tool. *Toolname* can be:

`dump`

Dumps the events from a SINC log file in a human-readable text format. Operates on a single SINC log file.

`diff`

Compares events in two SINC log files, noting any differences. Requires two SINC log files to be specified.

`analyze`

Analyzes a SINC log file for events that look suspicious. Operates on a single SINC log file.

`prune`

Trims a SINC log file down to the minimum number of events needed to preserve correct queue state. This can sometimes dramatically reduce SINC log file size without any loss of state. SINC log files are not pruned during normal operation due to performance considerations.

This tool operates on a single SINC log file, and outputs a new SINC log file in the same location and with the same name as the original plus an added ".pruned.n" suffix (where n is a counter starting at 1). The original SINC log file remains untouched. When pruning, we recommend to also specify the --config option, otherwise some state information may be lost in the pruned output.

The `-c` *configfile* or `--config=`*configfile* uses the given umestored XML config file. Specifying the umestored XML configuration file is optional, but can improve the accuracy of various tools, so its use is recommended.

The `-h` or `--help` displays this help and exits.

# 12. Configuration Reference for Umestored

The operating parameters for `umestored` come from an XML configuration file that must be supplied on the command line. `umestored` contains a **UM** context and receivers that may be configured with default values through a **UM** configuration file referenced in the XML configuration file. Default **UM** options my be overridden for each configured store using the XML configuration file.

You configure `umestored` to instantiate stores or queues with the `umestored` XML configuration file which **UM** reads at start up. This `umestored` XML configuration file has the following sections.

- Daemon section - holds administrative parameters for such things as the location of log files, the **UM** Configuration File, etc.
- Stores section - holds parameters for any persistent stores and also the topics to be persisted.
- Queues section - holds parameters for the queues to be instantiated along with the topics it should accept and the application set definitions which contain sets of receivers.

**High Level Stores Configuration File.**

```
<ume-store version="1.2">
  <daemon>
    Daemon configuration options
  </daemon>
```

```
  <stores>
    <store attributes>
      <topics>
        <topic attributes>
          <ume-attributes>
            <option attributes/>
          </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
</ume-store>
```

**High Level Stores and Queues Configuration File.** The stores and queues sections are both optional, although, you must specify at least one or the other.

```
<ume-store version="1.2">
  <daemon>
    Daemon configuration options
  </daemon>
  <stores>
    <store attributes>
      <topics>
        <topic attributes>
          <ume-attributes>
            <option attributes/>
          </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
  <queues>
    <queue attributes>
      <queue-groups>
        <queue-group attributes/>
      </queue-groups>
      <ume-attributes>
        <option attributes/>
      </ume-attributes>
      <indices>
        <index attributes>
          <ume-attributes>
            <option attributes/>
          </ume-attributes>
        </index>
      </indices>
      <application-sets>
        <application-set attributes>
          <ume-attributes>
            <option attributes/>
          </ume-attributes>
          <receiver-types>
```

```
            <receiver-type attributes>
              <ume-attributes>
                <option attributes/>
              </ume-attributes>
              <index-rules attributes>
                <rules>
              </index-rules>
            </receiver-type>
          </receiver-types>
        </application-set>
      </application-sets>
      <topics>
        <topic attributes>
          <ume-attributes>
            <option attributes/>
          </ume-attributes>
          <application-sets>
            </application-set attributes
          </application-sets>
        </topic>
      </topics>
    </queue>
  </queues>
</ume-store>
```

This section discusses the following topics.

- *Daemon Element*
- *Stores Element*
  - *Topics Element*
- *Queues Element*
  - *Queue Element*
  - *Indices Element*
  - *Application Sets Element*
  - *QueueTopics Element*
- *Option Types for ume-attributes Elements*
- *umestored Configuration DTD*
- *Store Configuration Example*
- *Queue Configuration Example*

## 12.1. Daemon Element

The following table presents child elements allowed in the *daemon* configuration section.

| Tag | Description | Default Value |
| --- | --- | --- |
| log | Required. Pathname for log file. | None--this is a required element. |
| pidfile | Pathname for daemon process ID (pid) file | No pidfile |
| uid | User ID (uid) for daemon process (if started as root) | Daemon retains starting uid |
| gid | Group ID (gid) for daemon process (if started as root) | Daemon retains starting gid |
| lbm-config | Pathname for **UM** configuration file | No config file; use **UM** defaults |
| lbm-license-file | Pathname for **UM** license file | License read from environment |
| web-monitor | Address:port where web monitor listens. Address of * listens on all interfaces. Also has a single attribute, **permission**, allowable values are *read-only* and *read-write*. Using *read-only* disables the text fields and buttons on a Web Monitor "debug page" that can only be enabled by Informatica Support. Example: <:web-monitor permission="read-only">*:15304<:/web-monitor> | No web monitor |
| lbm-password-file | Pathname for Queue Browser authentication file. | *rel-id/platform-id*/bin/password.xml |

## 12.2. Stores Element

The Stores Element is a container for individual store elements which define specific store instances. The below is an example of a Stores Element.

```
<stores>
  <store name="test-store-1" port="14567">
    <ume-attributes> ... </ume-attributes>
    <topics>
      <topic pattern="quote*" type="PCRE">
        <ume-attributes> ... </ume-attributes>
      </topic>
      <topic pattern="subject*" type="PCRE">
        <ume-attributes> ... </ume-attributes>
      </topic>
    </topics>
  </store>
  <store name="test-store-2" port="14568">
```

```
      <ume-attributes> ... </ume-attributes>
      <topics>
        <topic pattern="issue*" type="PCRE">
          <ume-attributes> ... </ume-attributes>
        </topic>
        <topic pattern="topic*" type="PCRE">
          <ume-attributes> ... </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
```

## 12.2.1. Store Element

The Store Element contains information about an individual **UMP** store and has attributes, options and topics. See the example below.

```
<store name="test-store-1" port="14567">
  <ume-attributes> ... </ume-attributes>
  <topics>
    <topic pattern="quote*" type="PCRE">
      <ume-attributes> ... </ume-attributes>
    </topic>
    <topic pattern="subject*" type="PCRE">
      <ume-attributes> ... </ume-attributes>
    </topic>
  </topics>
</store>
```

The following table gives attributes for *store* elements.

| Attribute | Description | Default Value |
|---|---|---|
| name | Specifies the name for the store. | None—this is a required attribute |
| port | TCP port where umestored should listen for this store. | None—this is a required attribute and a valid (non-zero) port must be specified. |
| interface | Specifies the network interface over which umestored accepts connection requests for this store. | 0.0.0.0 (INADDR_ANY) |

### *12.2.1.1. Child Elements of the Store Element*

The following table gives the child elements allowed in the *store* configuration section.

| Child Element | Description | Default Value |
|---|---|---|
| topics | A container for topic elements. See *Topics Element* for more information. | None |

| Child Element | Description | Default Value |
| --- | --- | --- |
| `ume-attributes` | A container for `option` elements. See *Options for a Store's ume-attributes Element* for more information. | None |

#### *12.2.1.2. Options for a Store's ume-attributes Element*

You can configure context (scope) options with a *`type`* attribute of *`lbm-context`*. UM passes such options through the normal receiver and context configuration option setting mechanisms. See the UM Configuration Guide (../Config/index.html) for details. Store options without a *`type`* attribute or those explicitly given a *`type`* attribute of *`store`* simply configure the store itself.

The following table gives options allowed for a *`store`* element. Use the `store` Option Type for these options. A Store's ume-attributes Element can also accept the `lbm-context` Option Type. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
| --- | --- | --- |
| `disk-cache-directory` | Pathname for disk store message cache directory | umestored-cache |
| `disk-state-directory` | Pathname for disk store state directory | umestored-state |
| `allow-proxy-source` | Allows the store to act as a proxy source in case a registered source terminates. | 0 (Disable) |
| `context-name` | Name of the store that can be used by sources to refer to the store instead of the address:request port. A store runs in its own context, therefore a name can used to represent the address:request port. This name facilitates **UMP** operation across the UM Gateway. Restricted to 128 characters in length, and may contain only alphanumeric characters, hyphens, and underscores. | None. |
| `retransmission-request-processing-rate` | Specifies the number of retransmission requests processed by a store per second across all topics. The store drops all retransmission requests that exceed this value. | 262144 |

### 12.2.2. Topics Element

The Topics element is a container element for all the topics persisted by the **UMP** store. It is one of the two child elements of the Store Element. See the example below.

```
<topics>
  <topic pattern="issue*" type="PCRE">
    <ume-attributes> ... </ume-attributes>
  </topic>
  <topic pattern="topic*" type="PCRE">
    <ume-attributes> ... </ume-attributes>
  </topic>
</topics>
```

#### 12.2.2.1. Topic Element

The Topic Element defines an individual topic persisted on the **UMP** store. The following table gives attributes for the *topic* element.

| Attribute | Description | Default Value |
|---|---|---|
| pattern | Specifies a pattern used to select topics for which a store provides persistence services. | None—this is a required attribute |
| type | Specifies the type of matching to be performed on the pattern attribute. A value of direct selects an exact string match. A value of PCRE selects a Perl Compatible Regular Expression match. A value of regexp selects a POSIX extended regular expression. PCRE, or regexp. | direct |

The Topic Element has one child element, *ume-attributes*, the options for which appear in *Options for a Topic's ume-attributes Element*.

##### 12.2.2.1.1. Options for a Topic's ume-attributes Element

The following table gives options allowed for a *topic* element. Use the store Option Type for these options. You can also configure receiver (scope) options and source (scope) options in a Topic's ume-attributes Element by using the Option Types lbm-receiver and lbm-source, respectively. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|
| retransmission-request-forwarding | If enabled (value = 1), the store always forwards retransmission requests to sources and does not service any requests itself. If disabled (value = 0), the store services retransmission requests for data it has, and forwards requests to sources for data it does not have. | 0 (store services retransmission requests) |

| Option | Description | Default Value |
|---|---|---|
| `repository-type` | Specifies how messages should be retained by the store. A value of *`no-cache`* does not retain messages, only state information. A value of *`memory`* retains messages only in the (presumably volatile) main memory of the store. A value of *`disk`* retains messages to (presumably non-volatile) disk storage as quickly as possible. In addition, messages are cached in main memory for a time as well. A value of *`reduced-fd`* retains messages in disk storage using significantly fewer File Descriptors. Use of this repository type may impact performance. (See *Persistent Store Architecture*.) The *`reduced-fd`* disk storage option is not available on Microsoft Windows. | no-cache |
| `repository-size-threshold` | For topics with a *`repository-type`* of *`memory`*, *`disk`* or *`reduced-fd`*, specifies the minimum number of message bytes (includes payload, headers, and store structure overhead) retained for a topic. For RPP repositories, this value only includes message payload. For the *`disk`* or *`reduced-fd`* repository type, this value configures the size threshold of the memory cache. (units: bytes) | 25165824 (24 MB) |
| `repository-size-limit` | For topics with a *`repository-type`* of *`memory`*, *`disk`* or *`reduced-fd`*, specifies the maximum number of message bytes (includes payload, headers, and store structure overhead) retained for each source. For RPP repositories, this value only includes message payload. For the *`disk`* or *`reduced-fd`* repository type, this value configures the size of the memory cache. (units: bytes) | 50331648 (48 MB) |

| Option | Description | Default Value |
|---|---|---|
| `repository-age-threshold` | For topics with a *`repository-type`* of *`memory`*, *`disk`* or *`reduced-fd`*, specifies a message age threshold. Younger messages will be retained. Space used to store older messages may be reclaimed. A value of 0 means message age is not considered in retention decisions. (units: seconds) | 0 |
| `repository-disk-max-async-cbs` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies the maximum number of outstanding async I/O callbacks for reading and writing messages to disk. (units: async callbacks) | 16 callbacks |
| `repository-disk-max-write-async-cbs` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies the maximum number of outstanding async I/O callbacks for writing messages to disk. Reducing this option can improve throughput by batching more fragments into a single write. (units: async callbacks) | 16 callbacks |
| `repository-disk-max-read-async-cbs` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies the maximum number of outstanding async I/O callbacks for reading messages from disk. Raising this value can improve recovery rates. For topics with a repository-type of *`reduced-fd`*, Informatica recommendeds a value of 200 times the number of expected receivers per topic. (units: async callbacks) | 16 callbacks |
| `repository-disk-file-size-limit` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies the maximum amount of disk space that will be used to store retained messages. A minimum value of 196992 is enforced. (units: bytes) | 104857600 (100 MB) |

| Option | Description | Default Value |
|---|---|---|
| `repository-disk-file-preallocate` | For topics with a *`repository-type`* of *`disk`*, If set to 1, **UMP** pre-allocates a store's cache files to match their maximum size on disk (as configured by `repository-disk-file-size-limit`) upon creation, as opposed to growing to that size as the store receives new messages. For ext3/4 and NTFS file systems, this options creates a sparse file, which does not allocate all of the underlying data blocks. Advantages of pre-allocation include better performance on rotating disks due to less file fragmentation, and knowing that enough disk space exists for any new source that registers. Disadvantage is the time to create the cache files, especially if many sources register at once. | 0 (zero) - do not pre-allocate |
| `repository-disk-async-buffer-length` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies the size of the buffers that will be used in async I/O operations for reading and writing messages to disk. A minimum value of 65664 is enforced. (units: bytes) | 65664 (64 KB + 128) |
| `repository-disk-message-checksum` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies whether the messages saved to disk should include a checksum field or not for validation if the store is restarted. (units: flag) | 0 (disabled) |

| **Option** | **Description** | **Default Value** |
|---|---|---|
| `source-activity-timeout` | Establishes the period of time from a source's last activity to the release of the source's RegID. Stores return an error to any new source requesting the source's RegID during this period. If proxy sources are enabled ( `ume_proxy_source` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEACTIVITYTIMEOUT)) the store does not release the source's RegID and **UMP** elects a proxy source. If neither proxy sources nor `ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMESTATELIFETIME) are configured, the store also deletes the source's state and cache. Can be overridden by `ume_activity_timeout` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEACTIVITYTIMEOUT). See also *Proxy Sources*. (units: milliseconds) | 30000 (30 seconds) |
| `source-state-lifetime` | Establishes the period of time from a source's last activity to the deletion of the source's state and cache by the store, regardless of whether a proxy source has been created or not. You can also configure `ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMESTATELIFETIME) for the source. The store uses whichever is shorter. See also *Proxy Sources*. (units: milliseconds) | 0 (zero) |
| `receiver-activity-timeout` | Establishes the period of time from a receiver's last activity to the release of the receiver's RegID. Stores return an error to any new request for the receiver's RegID during this period. Can be overridden by `ume_activity_timeout` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEACTIVITYTIMEOUT). See also *Proxy Sources*. (units: milliseconds) | 30000 (30 seconds) |

| Option | Description | Default Value |
|---|---|---|
| `receiver-state-lifetime` | Establishes the period of time from a receiver's last activity to the deletion of the receiver's state and cache by the store. You can also configure `ume_state_lifetime` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMESTATELIFETIME) for the receiver. The store uses whichever is shorter. See also *Proxy Sources*. (units: milliseconds) | 0 (zero) |
| `source-check-interval` | Specifies how often a store will check for activity of sources and receivers. (units: milliseconds) | 100 (100 milliseconds) |
| `keepalive-interval` | Specifies how often a store will generate keepalive traffic to sources and receivers if there has been no traffic required in the normal course of operation. (units: milliseconds) | 500 (500 milliseconds) |
| `receiver-new-registration-rollback` | Specifies the number of stabilized messages that a newly registered receiver should consume. For example, setting this to 10, "rolls back" the new receiver's starting message to the 10th most recent message. This value must be positive and less than 2147483648. The recommended value of 2147483647 indicates that the rollback should begin at the start of the stream. A value of 0 indicates the store should instruct the receivers to start with the next new message from the source known by the store. (units: messages) | 0 (no recovery requested) |
| `proxy-election-interval` | Specifies the interval, in milliseconds, used when electing a proxy source. When a source, which requested that a proxy source be provided for it, has been detected as no longer active, each store eligible to provide a proxy source for it waits for an amount of time which is randomized in the range [0.5*`proxy-election_interval` .. 1.5*`proxy-election_interval`]. If no other store has been elected to serve as the proxy source, the store declares itself as the proxy source. (units: milliseconds) | 5000 (5 seconds) |

| Option | Description | Default Value |
| --- | --- | --- |
| `stability-ack-interval` | Specifies the maximum amount of time that stability acknowledgments will be batched before being sent to a source. Batching stability ACKs can increase throughput of **UMP** stores (especially memory stores) significantly, but introduces a delay between when a message is actually stable in the **UMP** store and when the source is notified of message stability. (units: milliseconds) | 200 (200 milliseconds) |
| `stability-ack-minimum-number` | Specifies the minimum number of message stability acknowledgments that must accumulate before a stability ACK is sent to a source. With the default value of 1, stability ACKs are sent immediately as soon as messages are stable. Increasing this value causes stability ACKs to be batched, which can increase throughput of **UMP** stores (especially memory stores) significantly, but introduces a delay between when a message is actually stable in the **UMP** store and when the source is notified of message stability. If using a stability ACK-based flight size on a **UMP** source in combination with this option, it is advisable to make sure `stability-ack-minimum-number` is set less than or equal to the source's flight size. Otherwise, stability ACKs will only be sent upon expiration of the `stability-ack-interval` timer, resulting in bursty stop-and-go sending. (units: number of message fragments) | 1 (1 fragment) |

| Option | Description | Default Value |
|---|---|---|
| `repository-allow-receiver-paced-persistence` | Specifies if the repository allows receiver-paced persistence (1) or a source-paced persistence (0). The source must enable or disable this option with ume_receiver_paced_persistence (../Config/ultramessagingpersistenceoptions.html#SOURCEUMERECEIVERPACEDPERSISTENCE) but cannot enable it if this value is set to 0. | 0 (source-paced persistence) |
| `repository-allow-ack-on-reception` | Specifies if the repository should acknowledge to the source that it has persisted a message as soon as the repository receives it instead of when it writes the message to disk. Set this option to 1 to allow this behavior. The source must also set this option with `ume_repository_ack_on_reception` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEREPOSITORYACKONRECEPTION) to enable this behavior, but cannot enable it if this value is set to 0. For memory stores, this option has no effect. This option only applies to RPP repositories (`repository-allow-receiver-paced-persistence` = 1). | 0 (store acknowledges persistence when the store writes the message to disk) |
| `repository-disk-write-delay` | For topics with a *`repository-type`* of *`disk`* or *`reduced-fd`*, specifies the maximum delay in milliseconds before the repository persists a message to disk. If the repository sets this option a value other than the default, the source can reconfigure this option with `ume_write_delay` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEWRITEDELAY) to a lower or equal value, but cannot increase it. (units: milliseconds) | 0 milliseconds |

| Option | Description | Default Value |
|---|---|---|
| `source-flight-size-bytes-maximum` | Specifies the maximum message payload in bytes allowed to be inflight (un-stabilized at a store and without delivery confirmation) before a new message send either blocks or triggers a notification source event. **UMP** monitors both this option and `ume_flight_size` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZE). If either threshold is met, the configured blocking or notification behavior executes. See `ume_flight_size_behavior` (../Config/ultramessagingpersistenceoptions.html#SOURCEUMEFLIGHTSIZEBEHAVIOR). A source can reconfigure this option to a value of less than or equal to this value. This option only applies to RPP repositories (`repository-allow-receiver-paced-persistence` = 1). (units: bytes) | 4194304 bytes (4MB) |

## 12.3. Queues Element

The Queues Element is a container element for all queues. An abbreviated queues section appears below.

```
<queues>
  <queue name="Queue 1" interface=10.29.3.24" port="20555" group-index="0">
    <queue-groups>
      <queue-group index="0" size="5"/>
      <queue-group index="1" size="1"/>
    </queue-groups>
    <ume-attributes> ... </ume-attributes>
    <application-sets> ... </application-sets>
    <topics> ... </topics>
  </queue>
  <queue name="Queue 2" interface=10.29.3.25" port="20555" group-index="0">
    <queue-groups>
      <queue-group index="0" size="5"/>
      <queue-group index="1" size="1"/>
    </queue-groups>
    <ume-attributes> ... </ume-attributes>
    <application-sets> ... </application-sets>
    <topics> ... </topics>
  </queue>
```

```
</queues>
```

## 12.3.1. Queue Element

The Queue Element defines a single queue. Each queue must have a unique `name` and a `port`. An `interface`, `group-index` and `group-size` are optional. The following table gives attributes for a *`queue`* element.

| **Attribute** | **Description** | **Default Value** |
|---|---|---|
| `name` | Specifies the name for the queue. The queue name is the prime method for identifying the queue and must be unique. **UMQ** uses it during queue resolution, etc. | None—this is a required attribute |
| `port` | TCP port where `umestored` should listen for this queue. As with **UMP** stores, **UMQ** access the queue during registration and other operations with the port. | None—this is a required attribute |
| `interface` | Specifies the network interface over which `umestored` accepts connection requests for this queue. | 0.0.0.0 (`INADDR_ANY`) |
| `group-index` | A number that identifies this queue instance's queue group. See *Queue Redundancy* for more information. | 0 - Valid values range from 0 to 65535. |
| `group-size` | The number of queue instances in this queue's group. You can use this attribute to specify the number of queue instances instead of the *Queue Groups Element* if the queue consists of only one group. | 1 - Valid values range from 1 to 65535. |

The Queue Element can be configured with the *Queue Groups Element*, *Application Sets Element* and *QueueTopics Element*. The options for a Queue's *`ume-attributes`* appear in the following sections.

- *General Options for a Queue's ume-attributes Element*
- *Message Storage Options for a Queue's ume-attributes Element*
- *Queue Management Options for a Queue's ume-attributes Element*
- *Queue Slave Instance (QSI) Options for a Queue's ume-attributes Element*

### 12.3.1.1. Queue Groups Element

The *`queue-groups`* element contains *`queue-group`* elements that define all the queue groups that make up the queue. In the abbreviated Queues section shown in *Queues Element*, the *`queue`* element,

```
<queue name="Queue 1" interface=10.29.3.24" port="20555" group-index="0">
```

specifies `Queue 1` as a Queue Instance in Group 0. The *`queue-groups`* element in the same sample specifies that `Queue 1` comprises two groups, Group 0, which has 5 queue instances and Group 1, which has 1 queue instance.

```
<queue-groups>
  <queue-group index="0" size="5"/>
  <queue-group index="1" size="1"/>
</queue-groups>
```

To completely configure `Queue 1`, you must specify 6 queue instances in either one `umestored.xml` file as individual queue elements within the queues element or in separate `umestored.xml` files, one for each instance. The Queue Element for all 6 queue instances would be the same except for the *`interface`*, *`port`* and *`group-index`* because every instance of `Queue 1` must have the same *`name`*. And all 6 queue instances would also have the same *`queue-groups`* configured.

| Attribute | Description | Default Value |
| --- | --- | --- |
| `index` | The queue's redundancy group. See *Queue Redundancy* for more information. | 0 - Valid values range from 0 to 65535. |
| `size` | The number of queue instances in this queue's redundancy group. | 1 - Valid values range from 1 to 65535. |

### *12.3.1.2. General Options for a Queue's ume-attributes Element*

The table below displays the general options available for a Queue Element. Use the `queue` Option Type for these options. A Queue's ume-attributes Element can also accept Option Types `lbm-receiver`, `lbm-context` and `lbm-source`. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
| --- | --- | --- |
| `control-topic-name` | The name of the queue's control topic. The queue sends all control information for *Parallel Queue Dissemination (PQD)* and *Source Dissemination (SD)* to this topic. | `UMQ-cntl:qname` where `qname` is the name of the queue. |
| `queue-advertisement-interval` | The interval in milliseconds between advertisements of the queue for queue resolution. | 1000 (1.0 seconds). Value must be greater than 0. |
| `forwarding-behavior` | How **UMQ** forwards messages in the *Serial Queue Dissemination (SQD)* and *Parallel Queue Dissemination (PQD)* dissemination models. Valid values are `store-while-forward` and `store-then-forward`. | `store-while-forward` |

| Option | Description | Default Value |
|---|---|---|
| `dissemination-model` | The dissemination model used by the queue. Valid values are: `serial-queue-dissemination`, `SQD`, `parallel-queue-dissemination`, `PQD`, `source-dissemination`, and `SD`. See also *Message Paths* | `SQD` |
| `sending-threads` | The number of threads used to send control and data messages from the queue. | 1 (This must be greater than 0.) |
| `context-check-interval` | The interval in milliseconds between activity checks by the queue of a source or receiver context. Each check looks at the context as well as any associated source and receiver activity. | 1000 (1.0) seconds This must be greater than 0. |
| `context-activity-timeout` | The length of time a source or receiver context may be inactive before the queue disengages from the context. | 30000 (30.0 seconds) This must be greater than 0. |
| `context-keepalive-interval` | The interval in milliseconds between keepalives messages sent by the queue to determine whether a source or receiver context is alive or not. | 1000 (1.0 seconds) This must be greater than 0. |
| `source-activity-timeout` | The length of time a source may be inactive before the queue disengages from the source. | 30000 (30.0 seconds) This must be greater than 0. |
| `source-keepalive-interval` | The interval in milliseconds between keepalives messages sent by the queue to determine whether a source is alive or not. | 1000 (1.0 seconds) This must be greater than 0. |
| `rcr-save-timeout` | The maximum time in milliseconds that the queue holds RCR information for retransmission to receivers or other queue instances. | 30000 (30.0 seconds) This must be greater than 0. |
| `dead-letter-topic-prefix` | Prefix to be used for Dead Letter topic name. **UMQ** creates a Dead Letter topic name composed of this prefix, the unconsumed message's topic name or string and the Application Set name. See also *Dead Letter Queue*. | None. |

| Option | Description | Default Value |
|---|---|---|
| `dead-letter-topic-separator` | The separator **UMQ** uses between the `dead-letter-topic-prefix`, topic name or string and the Application Set name when it constructs the Dead Letter topic name. See also *Dead Letter Queue*. | / (forward slash) |
| `allow-browsing` | Allow observer receivers to retrieve lists of messages in the queue and to retrieve specific messages. Allowing browsing introduces some CPU and memory overhead within the queue, so if browsing support is not needed, setting this to 0 may increase performance. | 1 (browsing enabled). This must be 0 or 1. |
| `require-client-authentication` | Certain actions (currently topic list, message list, and message retrieve actions by default) require greater access permissions than a default queue user is given. Normally, the queue requires the client that is requesting the queue to perform any of those actions to pass a once-per-session authentication challenge by providing the username and password credentials of a user authorized to perform the requested action. If the client fails the authentication challenge, an error is returned to the client by the queue and the action is not performed. Turning this option off causes the queue to respond to such requests *even if* the user fails the authentication challenge. It is therefore NOT recommended to turn this option off in production environments, as it will make the queue less secure, but it is provided for convenience in functional test or development environments where security is not needed. | 1 (authentication required). This must be 0 or 1. |

#### *12.3.1.3. Message Storage Options for a Queue's ume-attributes Element*

**UMQ** provides 3 basic modes or operation for message storage.

- **Memory Storage**: **UMQ** stores messages and message state only in memory. A restarted `umestored` does not resume any previous operation. In this configuration, `sinc-data-filename`, `sinc-queue-swap-filename`, and `sinc-log-filename` are not set.
- **Disk Storage, no Persistence**: **UMQ** swaps messages and message state from memory to disk as needed to maintain low memory overhead. A restarted `umestored` does not resume any previous operation. In this configuration, `sinc-data-filename` and `sinc-queue-swap-filename` are set. The `sinc-log-filename` is not set.
- **Disk Storage, Persistence**: **UMQ** swaps messages and message state from memory to disk as needed to maintain low memory overhead. In addition, **UMQ** maintains a separate log file that contains all operations needed to reconstruct state when **UMQ** restores `umestored`. In this configuration, `sinc-data-filename`, `sinc-queue-swap-filename`, and `sinc-log-filename` are set.

Use the `queue` Option Type for these options. A Queue's ume-attributes Element can also accept Option Types `lbm-receiver`, `lbm-context` and `lbm-source`. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|
| `sinc-data-filename` | The filename used to store message data on disk. | None. |
| `sinc-queue-swap-filename` | The filename used to store message state for the queue. | None. |
| `sinc-log-filename` | The filename used to contain the queue log used to reconstruct state upon restarts. You can use the umqsltool utility to manage this file. | None. |
| `sinc-max-size` | The storage area's maximum size in bytes. When exceeded, the queue drops new messages until more space room is available. For file-based storage areas, the value of this option is bytes on disk. For memory-based storage, the value of this option is bytes of memory. | 104857600 bytes |
| `sinc-data-swap-threshold` | The threshold, in bytes, where enough message data causes the oldest and least referred to messages to be swapped to disk and removed from memory. | 104857600 bytes. This option must be greater than 0. |
| `sinc-block-swap-threshold` | The threshold, in blocks, where enough message state information causes the oldest and least referred to information to be swapped to disk and removed from memory. A block is state for roughly 120 messages for the same topic and Application Set and is roughly 4KB in size. | 128 blocks This option must be greater than 0. |

| **Option** | **Description** | **Default Value** |
| --- | --- | --- |
| `sinc-data-readahead` | The number of messages that are read in ahead of the current point in an Application Set's queue position. This helps to initiate asynchronous reads ahead of the time when they are needed for message assignment. | 16 |

### *12.3.1.4. Queue Management Options for a Queue's ume-attributes Element*

The Queue Management options control how queue instances communicate regarding elections and how they monitor each other.

Use the `queue` Option Type for these options. A Queue's ume-attributes Element can also accept Option Types `lbm-receiver`, `lbm-context` and `lbm-source`. See *Option Types for ume-attributes Elements* for more information.

| **Option** | **Description** | **Default Value** |
| --- | --- | --- |
| `queue-management-election-backoff-timeout` | The period of time, in milliseconds, before the Queue Election Master (QEM) recalls an election. This backoff is initiated only if an error occurs during the election operation or multiple election calls appear. To avoid livelock, the actual timeout value used is in the range [0.5 * timeout, 1.5 * timeout]. | 5000 (5.0 seconds) This option must be greater than 0. |
| `queue-management-election-bias` | The bias used during elections to determine the Post Election Master (PEM). A higher value biases the election to that queue instance. | 0 - This option must be less than or equal to 1000. |
| `queue-management-election-call-interval` | The interval, in milliseconds, between Election Calls by the Queue Election Master (QEM) once an election is called due to an unresponsive master. | 500 (0.5 seconds) This option must be greater than 0. |
| `queue-management-election-call-timeout` | The period of time, in milliseconds, that an election call can take. If the Queue Election Master (QEM) can not end the election early, then this timeout signals the end of the election and causes the QEM to generate a New Instance List (NIL). | 5000 (5.0 seconds) This option must be greater than 0. |
| `queue-management-join-request-interval` | The interval, in milliseconds, between Join Requests sent by queues that start up. | 500 (0.5 seconds) This option must be greater than 0. |

| Option | Description | Default Value |
|---|---|---|
| `queue-management-join-request-timeout` | The period of time, in milliseconds, after which a starting queue instance gives up on joining an existing queue and starts its own and elects itself master. | 5000 (5.0 seconds) This option must be greater than 0. |
| `queue-management-master-activity-timeout` | The period of time, in milliseconds, that a master must be active (by sending Queue Management Instance Lists) or be declared dead. A new Queue Election Master (QEM) calls an election after the expiration of this period. | 5000 (5.0 seconds) This option must be greater than 0. |
| `queue-management-master-check-interval` | The interval, in milliseconds, between checks on the master by the slaves. | 1000 (1.0 second) This option must be greater than 0. |
| `queue-management-new-instance-list-interval` | The interval, in milliseconds, at which the Queue Election Master (QEM) retransmits the New Instance List (NIL) after an election call has been completed. | 200 (0.2 seconds) This option must be greater than 0. |
| `queue-management-new-instance-list-timeout` | The period, in milliseconds, by which a Queue Election Master (QEM) must receive confirmations of the New Instance List (NIL) that it has sent out. If insufficient confirmations come back with this period expires, the QEM calls a new election and resets the timeout. | 2000 (2.0 seconds) This option must be greater than 0. |
| `queue-management-resume-operation-interval` | The interval, in milliseconds, at which a Queue Election Master (QEM) sends a resume operation message to the new master after the New Instance List (NIL) has been confirmed. | 200 (0.2 seconds)This option must be greater than 0. |
| `queue-management-resume-operation-timeout` | The timeout, in milliseconds, by which a Queue Election Master (QEM) must detect the resumption of operation after the New Instance List (NIL) has been confirmed. If resumption has not occurred when this timeout expires, the QEM calls a new election and resets the timeout. | 2000 (2.0 seconds) This option must be greater than 0. |
| `queue-management-slave-activity-timeout` | The period of time, in milliseconds, that a slave must be unresponsive before the master removes the slave from the Queue Management Instance List. | 5000 (5.0 seconds) This option must be greater than 0. |

| Option | Description | Default Value |
|---|---|---|
| `queue-management-slave-check-interval` | The interval, in milliseconds, between checks of the slaves by the master. This also controls how often the master sends Queue Management Instance List messages out to all the slaves. | 1000 (1.0 second) This option must be greater than 0. |

#### *12.3.1.5. Queue Slave Instance (QSI) Options for a Queue's ume-attributes Element*

Queue Slave Instances (QSIs) receive assignment information from the master queue using the control topic. These messages are called Receiver Control Records (RCR). If a queue instance misses an RCR, it can request it. The options below control the operation of requesting RCRs.

The acronym, `qrcrr`, refers to Queue Receiver Control Records Request.

Use the `queue` Option Type for these options. A Queue's ume-attributes Element can also accept Option Types `lbm-receiver`, `lbm-context` and `lbm-source`. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|
| `slave-qrcrr-generation-interval` | The timeout, in milliseconds, before an RCR must be retransmitted before **UMQ** abandons the request. | 10000 (10.0 seconds) This option must be greater than 0. |
| `slave-qrcrr-interval` | The interval, in milliseconds, between sending RCR retransmit requests. | 200 (0.2 seconds) This option must be greater than 0. |
| `slave-qrcrr-outstanding-maximum` | The maximum number of outstanding Receiver Control Records (RCR) to request at a single time. | 100 This option must be greater than 0. |
| `source-missing-message-timeout` | Controls the period of time that **UMQ** instance tracks source messages that are missed before abandoning them. | 5000 (5.0 seconds) This option must be greater than 0. |
| `source-save-timeout` | The timeout, in milliseconds, that the queue instance maintains historical state for a source that has had all its messages consumed. | 600000 (10.0 minutes) This option must be greater than 0. |

### 12.3.2. Indices Element

The Indices element is a container element for a queue's indices. (See *Indexed Queuing*.) Configuration of indices is optional, however, specifically configured indices can have rules (allow or deny) applied to them on a per receiver type basis. See *Index Rules Element*. Messages sent to the queue with a specifically configured index can be treated with different attributes than a normal message or a message sent on a non specifically-configured index. In the abbreviated sample below, *Red Messages* is an example of a named index, and *Brown Messages* is an example of a ranged numeric index.

```
<indices>
  <index name="Red Messages" type="named" value="red">
    <ume-attributes> ... </ume-attributes>
  </index>
  <index name="Brown Messages" type="ranged" value="13, 44, 80-90">
    <ume-attributes> ... </ume-attributes>
  </index>
</indices>
```

### 12.3.3. Index Element

The Index element defines a single index and has three attributes, a `name`, `type` and `value`. See the following examples:

```
<index name="Red Messages" type="named" value="red"> ... </index>
<index name="Brown Messages" type="ranged" value="4"> ... </index>
<index name="Yellow Messages" type="ranged" value="-10, 30-40, 90, 400-"> ... </index>
```

| Attribute | Description | Default Value |
| --- | --- | --- |
| `name` | Name of the index that also accompanies the message ( `lbm_umq_index_info_t` (../API/structlbm__umq__index__info__t__stct.html)). Can be a string or an unsigned 64-bit integer. | None. |
| `type` | An index type can be either the name of a single index or a range of indices. A `named` index is simply a string literal no longer than 216 characters. A `ranged` index can specify several ranges of unsigned 64-bit integers at once, or individual integers, separated by commas. A dash (`-`) can be used to indicate "and lower" or "and up". | None. |
| `value` | Either the index's string value (if a named index) or the index's range(s) of 64-bit unsigned numbers (if a ranged index). | None. |

Note that negative ranged indices are not allowed; the "`-10`" value in the example `Yellow Messages` index is shorthand for "`0-10`". The notation "`400-`" indicates "`400` and up", "up" meaning up to the maximum 64-bit unsigned integer, so it is shorthand for "`400-18446744073709551615`".

### 12.3.4. Options for an Index's ume-attributes Element

Options for an index appear in the following table. Use the `queue` Option Type for these options. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|
| `log-audit-trail` | Flag indicates whether or not the Queue logs receiver activity (assignments, consumption reports, reassignments, etc.) to the `umestored` log. Valid values are 0 (no logging) or 1. | 0 (zero) |
| `message-lifetime` | The maximum lifetime of a queue message in milliseconds. A value of 0 (zero) configures no lifetime. | 0 (zero) |
| `message-reassignment-timeout` | The maximum amount of time, starting from assignment, that a message may remain unconsumed by its assigned receiver before the Queue reassigns it to another receiver. A value of 0 (zero) configures the queue to never reassign. | 10000 (10.0 seconds) |
| `message-max-reassignments` | The maximum number of re-assignments allowed per message. **UMQ** applies the initial assignment to this maximum. The queue discards messages that exceed this maximum. Setting this option to 1 means that the message will never be reassigned. A value of 0 (zero) configures no maximum. | 0 (zero) |
| `receiver-activity-timeout` | The length of time a receiver may be inactive before the Queue disengages from the receiver. | 30000 (30.0 seconds) This must be greater than 0. |
| `receiver-keepalive-interval` | The interval in milliseconds between keepalive messages sent by the Queue to determine whether a receiver is alive or not. | 1000 (1.0 seconds) This must be greater than 0. |

### 12.3.5. Application Sets Element

The Application Sets element is a container element for every application set serviced by the queue. In the abbreviated sample below, *Set 1* is one application set and *Set 2* is another application set. Receiver Type ID of 100 identifies receivers in *Set 1* and Receiver Type ID of 200 identifies receivers in *Set 2*.

```
<application-sets>
```

```
  <application-set name="Set 1">
    <ume-attributes> ... </ume-attributes>
    <receiver-types>
      <receiver-type id="100">
          <ume-attributes> ... </ume-attributes>
      </receiver-type>
    </receiver-types>
  </application-set>
  <application-set name="Set 2">
    <receiver-types>
        <receiver-type id="200"/>
    </receiver-types>
  </application-set>
</application-sets>
```

#### 12.3.5.1. Application Set Element

The Application Set element defines an individual application set and has only one attribute, `name`, which identifies the application set. Each application set also requires a Receiver Type. See *Options for an Application Set's ume-attributes Element* for application set options.

#### 12.3.5.2. Options for an Application Set's ume-attributes Element

Options for an application set appear in the following table. Use the `queue` Option Type for these options. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|
| `log-audit-trail` | Flag indicates whether or not receiver activity (assignments, consumption reports, reassignments, etc.) are logged to the `umestored` log. Valid values are 0 (no logging) or 1. | 0 (zero) |
| `message-lifetime` | The maximum lifetime of a queue message in milliseconds. A value of 0 (zero) configures no lifetime. | 0 (zero) |
| `message-reassignment-timeout` | The maximum amount of time, starting from assignment, that a message may remain unconsumed by its assigned receiver before the queue reassigns it to another receiver. | 10000 (10.0 seconds) |

| Option | Description | Default Value |
|---|---|---|
| `message-max-reassignments` | The maximum number of re-assignments allowed per message. **UMQ** applies the initial assignment to this maximum. The queue discards messages that exceed this maximum. Setting this option to 1 means that the message will never be reassigned. A value of 0 (zero) configures no maximum. | 0 (zero) |
| `receiver-activity-timeout` | The length of time a receiver may be inactive before the queue disengages from the receiver. | 30000 (30.0 seconds) This must be greater than 0. |
| `receiver-keepalive-interval` | The interval in milliseconds between keepalive messages sent by the queue to determine whether a receiver is alive or not. | 1000 (1.0 seconds) This must be greater than 0. |
| `discard-behavior` | Queue behavior for unconsumed messages of the Application Set that have exceeded their `message-lifetime`. If set to `dead-letter`, **UMQ** places messages that exceed their `message-lifetime` on the Dead Letter Queue. If set to `drop`, **UMQ** discards such unconsumed messages. See also *Dead Letter Queue*. | `drop` |

#### *12.3.5.3. Receiver Types Element*

The Receiver Types element is a container element for all the receiver types within the Application Set.

##### *12.3.5.3.1. Receiver Type Element*

The Receiver Type element defines an individual receiver type and has only one attribute, `id`, which identifies the receiver type. A receiver type `id` is a 32-bit integer. See *Options for a Receiver Type's ume-attributes Element* for receiver type options.

##### *12.3.5.3.2. Options for a Receiver Type's ume-attributes Element*

Options for a Receiver Type appear in the following table. Use the `queue` Option Type for these options. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|

| Option | Description | Default Value |
| --- | --- | --- |
| `priority` | The priority assigned to the receiver type when assigning messages. Lower values have higher priority. This value may be negative. | 0 (zero) |
| `portion` | The maximum number of messages in flight for the receiver type. | 1 |

#### *12.3.5.3.3. Index Rules Element*

The Index Rules Element is a container element for index rule elements that can allow or deny receivers of the Receiver Type ID to process messages sent on specific configured indices. It contains one attribute, order, which specifies the order in which the Queue applies the index rules. Additionally, the `order` attribute specifies the *default* rule (allow or deny) to apply to an index for a particular receiver type if it is not specifically configured. Valid orders are either `"allow, deny"` or `"deny, allow"`.

- The Queue first applies rules of the first type specified in the `order` attribute.
- The Queue then applies rules of the second type specified and overrides any overlapping rules of the first type for the indices in which they overlap.
- The Queue then applies the default rule to any indices not specifically configured.

The example below establishes the following rules for receivers with Receiver Type ID 100.

1. Allowed to process messages with any of the Red Messages indices.
2. Explicitly prohibited from being assigned messages with any of the Brown Messages indices.
3. Prohibited (via the default rule) from being assigned messages sent with any other indices that fall outside those defined in either Red Messages or Brown Messages.

```
<receiver-type id="100">
  <ume-attributes> ... </ume-attributes>
  <index-rules order="allow, deny">
    <index-allow name="Red Messages" />
    <index-deny name="Brown Messages" />
  </index-rules>
</receiver-type>
```

> **Note:** The Index Rules Element is not available to ULB receivers.

## 12.3.6. QueueTopics Element

The Topics Element is a container element for all topics handled by the queue. In the abbreviated example below, application set *Set 1* is associated with the configured topic. Since the topic is a PCRE regular expression, any topic message submitted to the queue that matches the pattern is sent to *Set 1*.

```
<topics>
  <topic pattern="." type="PCRE">
    <ume-attributes> ... </ume-attributes>
      <application-sets>
        <application-set name="Set 1"/>
    </application-sets>
  </topic>
</topics>
```

### 12.3.6.1. Queue Topic Element

The Topic Element for a queue uses the same attributes for store topics listed in *Topic Element* and must also specify an Application Set that is configured in the Application Sets section. The options listed in *Options for a Queue Topic's ume-attributes Element* are also available.

### 12.3.6.2. Options for a Queue Topic's ume-attributes Element

Options for a queue topic appear in the following table. Use the `queue` Option Type for these options. A Queue's ume-attributes Element can also accept Option Types `lbm-receiver`, `lbm-context` and `lbm-source`. See *Option Types for ume-attributes Elements* for more information.

| Option | Description | Default Value |
|---|---|---|
| `message-management-check-interval` | The interval in milliseconds between status checks of the outstanding assignments for all assigned messages on the topic. These checks are done for lifetime and re-assignment. | 1000 (1.0 seconds) This value must not be 0. |
| `topic-check-interval` | The interval in milliseconds between checks for any sources and receivers of the topic. If none exist, the queue disengages from the topic. | 5000 (5.0 seconds) This value must not be 0. |
| `rcr-save-timeout` | The maximum time in milliseconds that the queue holds RCR information for retransmission to receivers or other queue instances. | 30000 (30.0 seconds) This must be greater than 0. |
| `message-dissemination-hold-interval` | The interval in milliseconds between dissemination of the same message to multiple receivers. | 10000 (10.0 seconds) This must be greater than 0. |

| Option | Description | Default Value |
|---|---|---|
| `dead-letter-topic` | Setting this option to **1** specifies this topic as a Dead Letter Topic to be used by **UMQ** to receive unconsumed messages of the Queue. (The `pattern` for this topic should match the `dead-letter-topic-prefix`.) This topic can only be configured with one Application Set that has its `discard-behavior` set to `drop` to prevent the chaining of Dead Letter Topics. See also *Dead Letter Queue*. | 0 (zero) |
| `message-total-lifetime` | Establishes the period of time from when a queue enqueues a message until the time the message cannot be assigned or reassigned to a receiver. The queue deletes the message upon expiration of the lifetime. See also *Message Lifetimes*. | 0 (zero) |

## 12.4. Option Types for ume-attributes Elements

All options configured for *`ume-attributes`* require an Option Type. The following table describes the five Option Types.

| Option Type | Description | Default Value |
|---|---|---|
| `lbm-receiver` | Allows you to configure receiver (scope) options that you usually specify in a **UM** Configuration File or set using `lbm_*_attr_setopt()`. This Option Type is appropriate for a Topic Element's *`ume-attributes`* child element. For example, you could turn off delivery of NAKs for a particular Topic by including the following within the Topic's *`ume-attributes`* element: `<option type="lbm-receiver" name="transport_lbtrm_send_naks" value="0">` . | None - this is a required attribute. |

| Option Type | Description | Default Value |
|---|---|---|
| `lbm-context` | Allows you to configure context (scope) options that you usually specify in a **UM** Configuration File or set using `lbm_*_attr_setopt()`. This Option Type is appropriate for a Store Element's *`ume-attributes`* child element. For example, you could increase the receiver socket buffer by including the following within the *`ume-attributes`* element: `<option type="lbm-context" name="transport_lbtrm_receiver_socket_buffer" value="1048576">` . | None - this is a required attribute. |
| `lbm-source` | Allows you to configure source (scope) options that you usually specify in a **UM** Configuration File or set using `lbm_*_attr_setopt()`. This Option Type is appropriate for a Topic Element's *`ume-attributes`* child element. For example, you could change the transport by including the following within the *`ume-attributes`* element: `<option type="lbm-source" name="transport" value="lbtru">` . | None - this is a required attribute. |
| `store` | Use this option type for all options that directly configure a store or repository and appear in either a *`store`* or *`topic`* *`ume-attributes`* element. For example, `<option type="store" name="context-name" value="remote-store"/>` or `<option type="store" name="repository-type" value="disk"/>`. | None - this is a required attribute. |
| `queue` | Option type used for all *`ume-attributes`* configured for the *`queue`* element and its *`application-set`*, *`receiver-type`* and *`topic`* elements. | None - this is a required attribute. |

## 12.5. umestored Configuration DTD

The DTD for version 1.2 of `umestored` appears below. See also the DTD revision table below.

| DTD Version | Release Date | Product Version | Supported Features |
|---|---|---|---|
| 1.0 | Feb. 2007 | **UME** 1.0 | Persistent Stores |
| 1.1 | April 2010 | **UME** 3.0.1 / **UMQ** 1.0 | Persistent Stores, Queues and Ultra Load Balancing (ULB) |
| 1.2 | March 2011 | **UME** 3.2 / **UMQ** 2.1 | Persistent Stores, Queues, Ultra Load Balancing (ULB), Dead Letter Queue, Indexed Queuing and Indexed ULB |

```
<!ELEMENT ume-store (daemon, stores?, queues?)>
<!ATTLIST ume-store version CDATA #REQUIRED>
<!ELEMENT daemon (log | uid | pidfile | gid | lbm-config | lbm-license-file | web-monitor)*>
<!ELEMENT log ( #PCDATA )>
<!ATTLIST log type CDATA #IMPLIED>
<!ATTLIST log xml:space (default | preserve) \"default\">
<!ELEMENT pidfile ( #PCDATA )>
<!ATTLIST pidfile xml:space (default | preserve) \"default\">
<!ELEMENT uid ( #PCDATA )>
<!ATTLIST uid xml:space (default | preserve) \"default\">
<!ELEMENT gid ( #PCDATA )>
<!ATTLIST gid xml:space (default | preserve) \"default\">
<!ELEMENT lbm-config ( #PCDATA )>
<!ATTLIST lbm-config xml:space (default | preserve) \"default\">
<!ELEMENT lbm-license-file ( #PCDATA )>
<!ATTLIST lbm-license-file xml:space (default | preserve) \"default\">
<!ELEMENT web-monitor ( #PCDATA )>
<!ATTLIST web-monitor xml:space (default | preserve) \"default\">
<!ATTLIST web-monitor permission CDATA #IMPLIED>
<!ELEMENT stores (store*)>
<!ELEMENT store (ume-attributes | topics)+>
<!ATTLIST store name CDATA #REQUIRED>
<!ATTLIST store interface CDATA #IMPLIED>
<!ATTLIST store port CDATA #REQUIRED>
<!ELEMENT topics (topic+)>
<!ELEMENT topic (ume-attributes | application-sets)*>
<!ATTLIST topic pattern CDATA #REQUIRED>
<!ATTLIST topic type (direct | PCRE | regexp) #IMPLIED>
<!ELEMENT ume-attributes (option+)>
<!ELEMENT option EMPTY>
<!ATTLIST option type (lbm-receiver | lbm-context | lbm-source | store | queue) #IMPLIED>
<!ATTLIST option name CDATA #REQUIRED>
<!ATTLIST option value CDATA #REQUIRED>
<!ELEMENT queues (queue*)>
<!ELEMENT queue (indices | application-sets | ume-attributes | topics | queue-groups)+>
<!ATTLIST queue name CDATA #REQUIRED>
<!ATTLIST queue interface CDATA #IMPLIED>
<!ATTLIST queue port CDATA #REQUIRED>
<!ATTLIST queue group-index CDATA #IMPLIED>
```

```
<!ATTLIST queue group-size CDATA #IMPLIED>
<!ELEMENT receiver-types (receiver-type+)>
<!ELEMENT indices (index+)>
<!ELEMENT index (ume-attributes)*>
<!ATTLIST index name CDATA #REQUIRED>
<!ATTLIST index type CDATA #IMPLIED>
<!ATTLIST index value CDATA #REQUIRED>
<!ELEMENT application-sets (application-set+)>
<!ELEMENT application-set (ume-attributes | receiver-types)*>
<!ATTLIST application-set name CDATA #REQUIRED>
<!ELEMENT receiver-type (ume-attributes*, index-rules?)>
<!ATTLIST receiver-type id CDATA #REQUIRED>
<!ELEMENT index-rules (index-allow | index-deny)*>
<!ATTLIST index-rules order CDATA #IMPLIED>
<!ELEMENT index-allow EMPTY>
<!ATTLIST index-allow name CDATA #REQUIRED>
<!ELEMENT index-deny EMPTY>
<!ATTLIST index-deny name CDATA #REQUIRED>
<!ELEMENT queue-groups (queue-group+)>
<!ELEMENT queue-group EMPTY>
<!ATTLIST queue-group index CDATA #REQUIRED>
<!ATTLIST queue-group size CDATA #REQUIRED>
```

## 12.6. Store Configuration Example

```
<?xml version="1.0"?>
<ume-store version="1.2">
  <daemon>
    <log>stored.log</log>
    <pidfile>stored.pid</pidfile>
    <web-monitor>*:15304</web-monitor>
  </daemon>

  <stores>
    <store name="test-store" port="14567">
      <ume-attributes>
        <option type="store" name="disk-cache-directory" value="cache"/>
        <option type="store" name="disk-state-directory" value="state"/>
        <option type="store" name="context-name" value="remote-store"/>
      </ume-attributes>
      <topics>
        <topic pattern="test*" type="PCRE">
          <ume-attributes>
            <option type="store" name="repository-type" value="disk"/>
            <option type="store" name="repository-size-threshold" value="104857600"/>
            <option type="store" name="repository-size-limit" value="209715200"/>
            <option type="store" name="repository-disk-file-size-limit" value="1073741824"/>
            <option type="store" name="source-activity-timeout" value="120000"/>
            <option type="store" name="receiver-activity-timeout" value="120000"/>
            <option type="store" name="retransmission-request-forwarding" value="0"/>
          </ume-attributes>
```

```
        </topic>
      </topics>
    </store>
  </stores>
</ume-store>
```

## 12.7. Queue Configuration Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ume-store version="1.2">
  <daemon>
    <log>/tmp/qlog.txt</log>
    <lbm-license-file>/home/.ume_license</lbm-license-file>
    <lbm-config>/home/queues/lbm-config.cfg</lbm-config>
    <web-monitor>*:20395</web-monitor>
  </daemon>
  <queues>
    <queue name="sample_queue" port="20555" group-index="0">
      <queue-groups>
        <queue-group index="0" size="5"/>
        <queue-group index="1" size="1"/>
      </queue-groups>
      <ume-attributes>
        <option type="queue" name="dissemination-model" value="SQD"/>
        <option type="queue" name="sinc-data-filename" value="/tmp/sample_queue_1-sd"/>
        <option type="queue" name="sinc-queue-swap-filename" value="/tmp/sample_queue_1-sqs"/>
        <option type="queue" name="sinc-log-filename" value="/tmp/sample_queue_1-sl"/>
        <option type="queue" name="sinc-max-size" value="25000000"/>
        <option type="queue" name="sending-threads" value="1"/>
        <option type="queue" name="sinc-block-swap-threshold" value="100000"/>
        <option type="queue" name="source-missing-message-timeout" value="5000"/>
        <option type="lbm-source" name="transport" value="lbtrm"/>
      </ume-attributes>
      <application-sets>
        <application-set name="Set 1">
          <ume-attributes>
            <option type="queue" name="log-audit-trail" value="0"/>
            <option type="queue" name="message-lifetime" value="6000"/>
            <option type="queue" name="message-reassignment-timeout" value="2000"/>
            <option type="queue" name="message-max-reassignments" value="10"/>
          </ume-attributes>
          <receiver-types>
            <receiver-type id="100">
              <ume-attributes>
                <option type="queue" name="priority" value="1"/>
                <option type="queue" name="portion" value="10"/>
              </ume-attributes>
            </receiver-type>
            <receiver-type id="101">
              <ume-attributes>
                <option type="queue" name="priority" value="100"/>
```

```
            <option type="queue" name="portion" value="10"/>
          </ume-attributes>
        </receiver-type>
      </receiver-types>
    </application-set>
    <application-set name="Set 2">
      <ume-attributes>
        <option type="queue" name="log-audit-trail" value="0"/>
        <option type="queue" name="message-lifetime" value="6000"/>
        <option type="queue" name="message-reassignment-timeout" value="2000"/>
        <option type="queue" name="message-max-reassignments" value="10"/>
      </ume-attributes>
      <receiver-types>
        <receiver-type id="200">
          <ume-attributes>
            <option type="queue" name="priority" value="1"/>
            <option type="queue" name="portion" value="10"/>
          </ume-attributes>
        </receiver-type>
        <receiver-type id="201">
          <ume-attributes>
            <option type="queue" name="priority" value="100"/>
            <option type="queue" name="portion" value="10"/>
          </ume-attributes>
        </receiver-type>
      </receiver-types>
    </application-set>
  </application-sets>
  <topics>
    <topic pattern="testA" type="PCRE">
      <application-sets>
        <application-set name="Set 1"/>
      </application-sets>
    </topic>
    <topic pattern="testB" type="PCRE">
      <application-sets>
        <application-set name="Set 2"/>
      </application-sets>
    </topic>
    <topic pattern="testC" type="PCRE">
      <application-sets>
        <application-set name="Set 1"/>
        <application-set name="Set 2"/>
      </application-sets>
    </topic>
  </topics>
</queue>
</queues>
</ume-store>
```

# 13. Ultra Messaging Web Monitor

The built-in web monitor (configured in the umestored XML configuration file) is a rich source of information about the health of a **UM** stores and queues. This section contains a page-by-page guide to reading and interpreting the output of a **UM** web monitor, with just a couple example sources and one receiver using a single store. This section discusses the following topics.


- *Ultra Messaging Web Monitor Index Page*
- *Persistent Stores Page*
- *Store Page*
- *Source Page*
- *Receiver Page*
- *Queue Page*
- *Queue Topic Page*


## 13.1. Ultra Messaging Web Monitor Index Page

The web monitor's index page tells what build of **UM** is running.

**Figure 24. UM Web Monitor Index Page**

Click on the link, Stores, to see the *Persistent Stores Page*.

## 13.2. Persistent Stores Page

**Figure 25. Persistent Stores Page**

This page shows all the stores configured under the `umestored` process. If you had 5 stores configured, they would be numbered Store 0 through Store 4. Our example has only one store configured, `ume-test-store`. Click on the link, ume-test-store, to see the *Store Page*.

## 13.3. Store Page

**Figure 26. Persistent Stores Page**

This page shows the following information about the store.

| Item | Description |
|---|---|
| Interface | This store is listening on all interfaces (0.0.0.0) on port 41394 |
| Cache Dir | Pathname for disk store message cache directory. This would be configured as a store attribute in the store's XML configuration file. `<option type="store" name="disk-cache-directory" value="cache/" />` |
| State Dir | Pathname for disk store state directory. This would be configured as a store attribute in the store's XML configuration file. `<option type="store" name="disk-state-directory" value="state/" />` |
| Configured Retransmission Request Processing Rate | Current value for the store's `retransmission-request-processing-rate` setting. |
| Retransmission Request Received Rate | Number of retransmission requests received per second. |
| Retransmission Request Service Rate | Number of retransmission requests serviced per second. |
| Retransmission Request Drop Rate | Number of retransmission requests dropped per second. Requests are dropped if the rate of retransmission requests exceeds the configured retransmission request rate. |
| Retransmission Request Total Dropped | The number of retransmission requests since the time the store was started. |
| Patterns | Specifies the wildcard pattern used to select topics for which a store will provide persistence services. This would be configured as a topic attribute in the store's XML configuration file. `<topic pattern="test*" type="PCRE">` |
| Topics | Displays the topic name and Registration IDs of the two sources publishing on the topic, 2369562861 and 3131255877. |

You can review information about the sources publishing on the topic by clicking on Registration ID displayed. The *Source Page* appears.

## 13.4. Source Page

**Figure 27. UM Web Monitor Source Page**

The following table explains the information found in the title of the Source Page.

| Source Page Title | Description |
|---|---|
| 2504558780 | The source's registration ID. |
| 10.29.3.42.14392 | The IP address and port of the source's **UM** configuration option, `request_tcp_port`. |
| 3958260924 | The source's transport session index. |
| 1161732811 | The source's topic index within the transport session, 3958260924. |

The transport session and topic indices are useful for debugging purposes when combined with a Wireshark capture, but are otherwise not relevant here. The following table provides descriptions of the items in the source page.

| Source Page Item | Description |
|---|---|
| Topic | **test** is the source's topic string. |
| Last Activity | **09:19:39.501350** is the timestamp when the store last heard from the source, including keepalives sent by **UM** |
| Repository | **disk** is the type of repository. |
| Receiver Paced Persistence | Setting for Receiver-paced Persistence (RPP), which is a repository option both the repositrory and source must enable. A value of 0 means RPP is not enabled and the repository is using the default Source-paced Resistence. A value of 1 means RPP is enabled. |
| Message Map: 104 | **104** is the total number of message fragments the store has for this source, both on disk and in memory. These are **UM**-level fragments, not IP-level fragments. **UM** messages are fragmented into roughly 8 kilobyte chunks for UDP-based protocols (LBT-RM and LBT-RU) and into roughly 64 kilobyte chunks for LBT-TCP. The majority of application messages tend to be well under the fragment boundaries, so the value after "Message Map" could be used as a rough estimate of the number of messages in the store from this particular source. It's at least a strict upper bound. |
| Window: [0, 0, 67] | The first **0** is the **trailing sequence number**, which is the oldest sequence number in the store for this source. In most cases, this starts at 0 and stays there for a while, but especially with **UME** 2.0 where stores can come and go, that may not be the case. It would also move if you, for example, hit a disk file size limit and had to throw out some old messages. |
| The second **0** is the **trailing sequence number** for messages **in memory**, so it is the oldest sequence number still in memory. Typically, you might have more sequence numbers on disk than you do in memory, or possibly the same number. | |

| Source Page Item | Description |
|---|---|
| The third number, **67**, is the **leading sequence number**, which is the highest sequence number in the store. | |
| ***NOTE:*** For a memory store, the first and second values would always be the same (the oldest sequence number in memory is the oldest in the store), so only two values are displayed; the trailing sequence number and the leading sequence number. These are sequence numbers of message fragments; there's usually just one fragment per message, but there could be more than one. | |
| Memory: 7176 / 52428800 / 104857600 | First number, **7176**, is the number of bytes of messages (which includes headers and a bit of store overhead) in memory. |
| The second number, **52428800**, is the `repository-size-threshold` topic option found in the store's XML configuration file. | |
| The third number, **104857600**, is the `repository-size-limit` setting. | |
| You would expect the number of bytes in memory to be under the threshold most of the time, but it could spike above it before going back down if the store is really busy momentarily. It should never go above the limit. | |
| Age Threshold: 0 | **0** is the `repository-age-threshold` setting. |
| Sync: [c2f, c2f, c2f] | Pertains to **disk** or **reduced-fd** repositories only. Sync format is: `sync_complete_sqn`, `sync_sqn`, `contig_sqn` |

| Source Page Item | Description |
| --- | --- |
| **sync_complete_sqn, c2f** Most recent sequence number that the Operating System has confirmed persisting to disk. | |
| **sync_sqn, c2f** Most recent sequence number for which the store has initiated persisting to disk, but the Operating System has not confirmed completion of persistence. | |
| **contig_sqn, c2f** Most recent sequence number that along with the trail_sqn, creates a range of sequence numbers with no sequence number gaps. For example, if `trail_sqn = 0` and the store has persisted all eleven messages with sequence numbers 0 through 10, `contig_sqn` would equal 10. `contig_sqn` would also be 10 if a receiver declared message sequence number 7 unrecoverably lost. contig_sqn would be 6 if message sequence number 7 was not persisted, but not declared lost. | |
| In progress: 0 / 0 | Pertains to **disk** or **reduced-fd** repositories only. In progress format is: `num_ios_pending` / `num_read_ios_pending` |
| **num_ios_pending, 0** Number of disk writes the store has submitted to the Operation System. A disk write refers to the store persisting a message to disk. | |

| Source Page Item | Description |
|---|---|
| The **num_read_ios_pending, 0** Number of disk reads that the store has submitted to the Operating System. A disk read, for example, results from an application retransmission request. | |
| Offsets: 0 / 190320 / 4294967296 | Pertains to **disk** or **reduced-fd** repositories only. Offsets format is: `start_offset`, `offset`, `max_offset` |
| **start_offset, 0** The relative location of the first message, `trail_sqn`, in the disk. `start_offset` is 0 for a **reduced-fd** repository. | |
| The **offset, 190320** The relative location of where the message, `contig_sqn` plus one will be written. `offset` represents the size of the repository on disk for a **reduced-fd** repository. | |
| **max_offset, 4294967296** The maximum size of the cache file. `max_offset` is the maximum repository size on disk for a **reduced-fd** repository. | |
| Active ULBs: 0 high 0 | **ULB** stands for **Unrecoverable Loss Burst**. A little extra work is required to keep cache files consistent when the store gets an unrecoverable loss burst, because unrecoverable loss bursts are delivered all at once for lots of messages, rather than one at a time like normal unrecoverable loss messages. |
| **Active ULB** is the number of unrecoverable loss burst events the store is dealing with at the moment. It'll go to zero after the ULB has been resolved. | |

| Source Page Item | Description |
|---|---|
| The **high** number (**0**) is the highest sequence number reported among any unrecoverable loss burst event, and is not reset after the ULB is handled; it increments throughout the process life of the store. | |
| ***WARNING:*** If you see any number other than 0 here, the store is losing large numbers of messages, and they are likely not being persisted. | |
| Loss: 0 ULBs 0 | These values are counters for number of unrecoverable loss messages (**Loss**) and for number of unrecoverable burst loss messages (**ULB**). These start at 0 when the store starts up and aren't reset until the store exits. They don't include any loss events that were persisted to disk from a previous run, only new loss events since the store started. There are cases with **UME** 2.0 where one individual store could legitimately report some unrecoverable loss, or maybe even unrecoverable loss bursts. |
| ***WARNING:*** If you see **any** number other than **0** for either of these counters, you should investigate. | |
| Drops: 0 / 0 | If the store is nearing the `repository-size-limit` and gets another message, the store will intentionally **drop** a message. A drop requires a bit of work on the store's part. |
| The first **0** is the number of **active drops**, which are drops that are currently being worked on. | |
| The second **0** is the total number of drops that have happened for this store since it was started. Some people want a low `repository-size-limit` and therefore lots of intentional drops can occur. Some don't want to drop any message the whole day - so the interpretation of the values is up to you. | |

| Source Page Item | Description |
|---|---|
| LBM Stats | These represent transport-level statistics for the underlying receivers in the store for the source. The example shown is for a TCP source, so not too many stats are available (stats for a TCP source are less important from a monitoring perspective). |
| Statistics for an LBT-RM or LBT-RU source, however, show number of NAKs sent, which is important. Ideally, the number of NAKs sent should be 0. A few NAKs from a store throughout the day is not an emergency. It can be, however, an early warning sign of more severe problems, and should be taken seriously. | |
| If you see a non-zero number of NAKs here, take a look at the overall network load the store's machine is attempting to handle, particularly in very busy periods and spikes; it may be too much. | |
| Receivers | Registration IDs for the receivers listening on the source's topic. You can review information about the receivers listening on the topic by clicking on Registration ID. The *Receiver Page* appears. |

## 13.5. Receiver Page

**Figure 28. UM Web Monitor Receiver Page**

The following table explains the information found in the title of the Receiver Page.

| Receiver Page Title | Description |
|---|---|
| 2504558781 | The receiver's registration ID. |
| 10.29.3.42.14393 | The IP address and port of the source's **UM** configuration option, `request_tcp_port`. |
| 1510613393 | The receiver's transport session index. |

| Receiver Page Title | Description |
|---|---|
| 1161732811 | The source's topic index within the transport session, 1510613393. |

The receiver page shows the following information.

| Receiver Page Item | Description |
|---|---|
| Topic | The topic that the receiver is listening on. |
| Last Activity | **09:09:35.981110** is the timestamp of when the store last heard from the receiver, including keepalives sent by **UM**. |
| Source RegID | Registration ID of the source publishing on the topic. Click on the Registration ID link to display the Source Page. |
| Source Session ID | The Session ID of the Source sending messages on the topic. |
| ACK | **c93** is the last message sequence number the receiver acknowledged. |

## 13.6. Queue Page

**Figure 29. Queue Page**

This page shows the following information about `Queue 0`, which is named **Queue 1**. The queue name is an attribute of the Queue Element in the Queue's XML configuration file. `<queue name="Queue 1" port="4567" group-index="0">`

| Item | Description |
|---|---|
| Interface | This queue is listening on all interfaces (0.0.0.0) on port 4567 |
| Queue ID | Identification given to this queue by **UMQ** . |
| Sending Threads | Number of threads configured for this queue to send control and data messages. This would be configured as a Queue Element option in the queue's XML configuration file. `<option type="queue" name="sending-threads" value="1/" />` |
| Sending Threads(s) Queue Size | The number of messages waiting to be sent to receivers in the pool of sending threads. This could be data and control messages for Parallel Queue Dissemination (PQD), data messages only for Serial Queue Dissemination (SQD) or control messages only for Source Dissemination (SD). |
| Registered Contexts | Number of application contexts registered with this queue. |
| Retransmit Requests | **Message Requests:** Number of requests for data message retransmission. |
| **Queue RCR Requests:** Number of requests for the retransmission of control information. | |

| Item | Description |
|---|---|
| **Dropped Requests:** Number of retransmission requests dropped by **Queue 1**. | |
| Patterns | Specifies the wildcard patterns used to select topics for which a queue will accept data messages. This would be configured as a topic attribute in the queue's XML configuration file. `<topic pattern="." type="PCRE">` |
| Topics | Displays the RCR Index (**697157a5**) and topic name (**a.b**) of the topic(s) configured for this queue. |

You can review information about the Queue's topics and application sets by clicking on the topic's RCR Index, **697157a5**). The *Queue Topic Page* appears.

## 13.7. Queue Topic Page

**Figure 30. Queue Topic Page**

This page shows the following information about the queue topic, **a.b**. This topic's RCR Index is **697157a5**.

| Item | Description |
|---|---|
| Queue | The Queue Name for which this topic is configured. It is also a link back to the Queue Page for this queue. |
| Application Sets | **Queue 1** has **2** Application Sets configured. |
| Consumed Messages | Total number of messages consumed by all Application Sets. |
| Reassignments | Number of messages that have been reassigned. |
| Topic RCR Requests | Number of requests for the retransmission of control information regarding this topic. |
| Saved RCRs | Number of Receiver Control Records save due to retransmissions. You configure how long the queue saves RCRs a Queue Element attribute in the queue's XML configuration file. `<option type="queue" name="rcr-save-timeout" value="30000"/>`. |
| Application Set | **Set 2** is the name of this Application Set. |
| **Enqueued Messages:** Number of messages currently held in **Queue 1** for **Set 2**. | |
| **Currently Assigned:** Number of messages currently assigned to a receiver in this Application Set. | |

| Item | Description |
|---|---|
| **Currently Reassigning:** Number of messages waiting to be reassigned to receivers. | |
| **Consumed Messages:** Number of messages consumed by receivers in this Application Set. | |
| **Reassignments:** Number of messages that have been reassigned to another receiver in this Application Set. | |
| **Discarded Messages:** Number of messages assigned to receivers in this Application Set that have been discarded. | |
| Receivers | Number of receivers (**1**) configured for this Application Set. Specific information for each receiver appears in the table below this item. |
| **ID:** The Assignment ID given to this receiver by the queue. | |
| **Address:** The address and port of the receiver. | |
| **Portion:** This receiver's portion size that you configure as a Receiver Type attribute in the queue's XML configuration file. `<option type="queue" name="portion" value="1"/>`. | |
| **Priority:** This receiver's priority that you configure as a Receiver Type attribute in the queue's XML configuration file. `<option type="queue" name="priority" value="1"/>`. | |

| Item | Description |
| --- | --- |
| **Outstanding:** Number of assigned messages for which the queue has not yet received Consumption Reports. | |
| **Last Active:** A timestamp indicating the last activity for the receiver. | |
| **Consumed:** Total messages consumed by this receiver. The total for this column should match the Consumed Messages value for the Application Set. | |
| Assigned | The number of messages currently assigned to all receivers for the Application Set. For each assigned message, the Message ID and Assignment ID for the receiver assigned the message appears. In addition, the **Reassign** and **Discard** links allow you to reassign or discard the individual message. |