Ultra Messaging (Version 6.12)

Guide for Persistence

Contents

Chapter 1

Introduction

This document describes the Persistence functionality of the UMP and UMQ products.

Attention

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

This document assumes familiarity with the UM Concepts Guide.

See **UM Glossary** for Ultra Messaging terminology, abbreviations, and acronyms.

Chapter 2

Persistence Overview

Ultra Messaging provides two different qualities of service (QOS) levels, related to likelihood of successful message delivery: streaming and persistence.

Streaming is the basic QOS level for UM. With Streaming, a published message will be delivered to a receiver reliably if the following requirements are met:

- the publisher and subscriber are both running,
- the message was published *after* the subscriber has had enough time to discover and join the publisher's data stream (note that UM's **Late Join** feature which somewhat relaxes this requirement), and
- the data link between the publisher and subscriber has a low-enough error rate that any lost data has time to be recovered by the Transport protocol within the time allotted for that recovery.

With Streaming, if a subscriber exits mid-stream (either intentionally or by failure), when that subscriber restarts, it typically cannot recover the messages that were sent during its absence.

The higher QOS available for UM is Persistence, by which messages can be delivered even in cases where one or more of the above requirements cannot be met. For example, if a publisher sends a message and then exits, and after that a subscriber starts, Persistence is required for that message to be delivered.

UM's Persistence functionality is implemented by components called "Stores" obtaining copies of published messages and writing them to non-volatile storage. These store components can be used by subscribers to recover messages that cannot be recovered by the Transport protocol. In particular:

- the publishers and subscribers to not have to be running at the same time,
- messages published prior to the subscriber joining the transport can be can be recovered, and
- even extended periods of data link failure can be recovered from after the data link is restored.

With Persistence, if a subscriber exits mid-stream (either intentionally or by failure), when that subscriber restarts, it will automatically recover the messages sent during its absence.

A system using UM 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.

Note

> The UMS product offers streaming QOS. The UMP and UMQ products offer both streaming and persistence QOS.

Chapter 3

Persistence Concepts

In discussing Persistence, 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. Persistence is not designed to address ongoing, corrupting agents. Rather, if one of its components fails, the design of Persistence enables it to continue supporting its ongoing operations at some level.

UM offers persistence in the following two modes:

- Source-paced Persistence (SPP) - default mode - the rate of message consumption by receivers does not constrain the rate a source can send. Persistent Stores write all messages to non-volatile storage, and messages are retained until they are overwritten when the allocated storage is filled. See Persistence Normal Operation.
- Receiver-paced Persistence (RPP) - optional mode - the rate of message consumption by receivers *does* constrain the rate a source can send. Persistent Stores only write message to non-volatile storage if one or more necessary receiver is slow in consuming the messages, and messages are deleted from the Store once all necessary receivers have consumed the message. See RPP Normal Operation.

3.1 Persistent Store Concept

UM uses a daemon to persist source and receiver state outside the actual sources and receivers themselves. This daemon is the 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. Persistent stores need not be a single entity. For fault tolerance purposes, it is possible to configure multiple stores in various ways.

For more information, see:

- Adding the Store to a Source,
- Designing Persistent Stores,
- Store Configuration Considerations,
- umestored Man Page,
- Configuration Reference for Umestored.

3.2 Registration Identifier Concept

UM persistence 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 UM enables your application to use and handle RegIDs very freely, you must use RegIDs carefully to avoid destructive results.

For more information, see:

- Adding Fault Recovery with Registration IDs
- Registration Identifiers
- RegIDs can also be managed easily with the use of Session IDs. See Managing RegIDs with Session IDs.

3.3 Delivery Confirmation Concept

A persistent receiver provides confirmation (acknowledgement) to the persistent store as it consumes (processes) messages. This is fundamental to the design of UM persistence.

The receiver can optionally provide this confirmation (acknowledgment) to the persistent source. These confirmations are turned off by default, but can be requested through either or both two configuration options:

- **ume_confirmed_delivery_notification (source)** - deliver a source event to the application indicating message consumption.
- **ume_retention_unique_confirmations (source)** - include receiver consumption as part of source flight size calculation.

These two options are unrelated to each other, except that they both request the receiver to send delivery confirmations. Note that when either or both of the options are set, the persistent source *requests* that the persistent receiver supply delivery confirmations. The persistent receiver has the option to decline the request by setting the option **ume_allow_confirmed_delivery (receiver)** to 0.

Note

Smart Sources do not support either form of delivery confirmation.

The latter option, **ume_retention_unique_confirmations (source)**, can provide a form of receiver-pacing; the source will not be allowed to exceed Persistence Flight Size beyond receiving applications. For more information, see: Confirmed Delivery

3.4 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, see Source Message Retention and Release.

3.5 Message Stability Concept

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, and those stores acknowledge stability to the sources. Since it takes time to write messages to disk and signal stability, the source is allowed to continue sending messages while waiting for stability acknowledgements. Any messages sent but not yet acknowledged are said to be "in flight". The number of in-flight messages is normally limited. For more information, see Persistence Flight Size.

In addition, UM informs the application when messages are stabilized. Until that stability acknowledgement is received, the source can not assume the messages will be successfully delivered. The message stability acknowledgement is vital to ensuring that messages will not be lost. For more information, see Source Message Retention and Release.

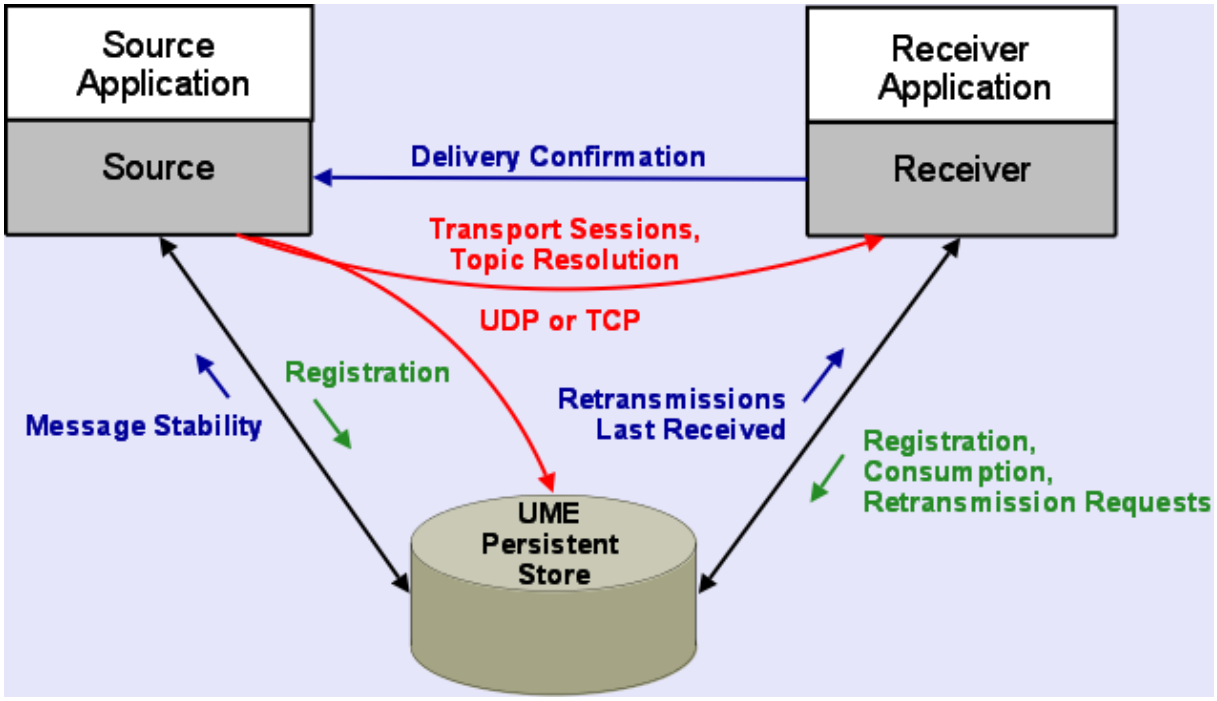
3.6 Quorum/Consensus Store Failover

Typically, multiple stores are deployed for simultaneous redundant operation. In this configuration, one or more stores (or the hosts they run on) can fail without impacting the message flow from sources to receivers, as long as a *quorum* of the configured stores is operational. UM defines a quorum as a majority of the configured stores. E.g. if 3 stores are configured, messaging can continue as long as at least 2 are operational. If 5 stores are configured, messaging can continue if at least 3 are operational. (Quorum/Consensus requires that an odd number of stores be configured.)

Chapter 4

Persistence Architecture

As shown in the diagram, UM provides messaging functionality as well as persistent operation.



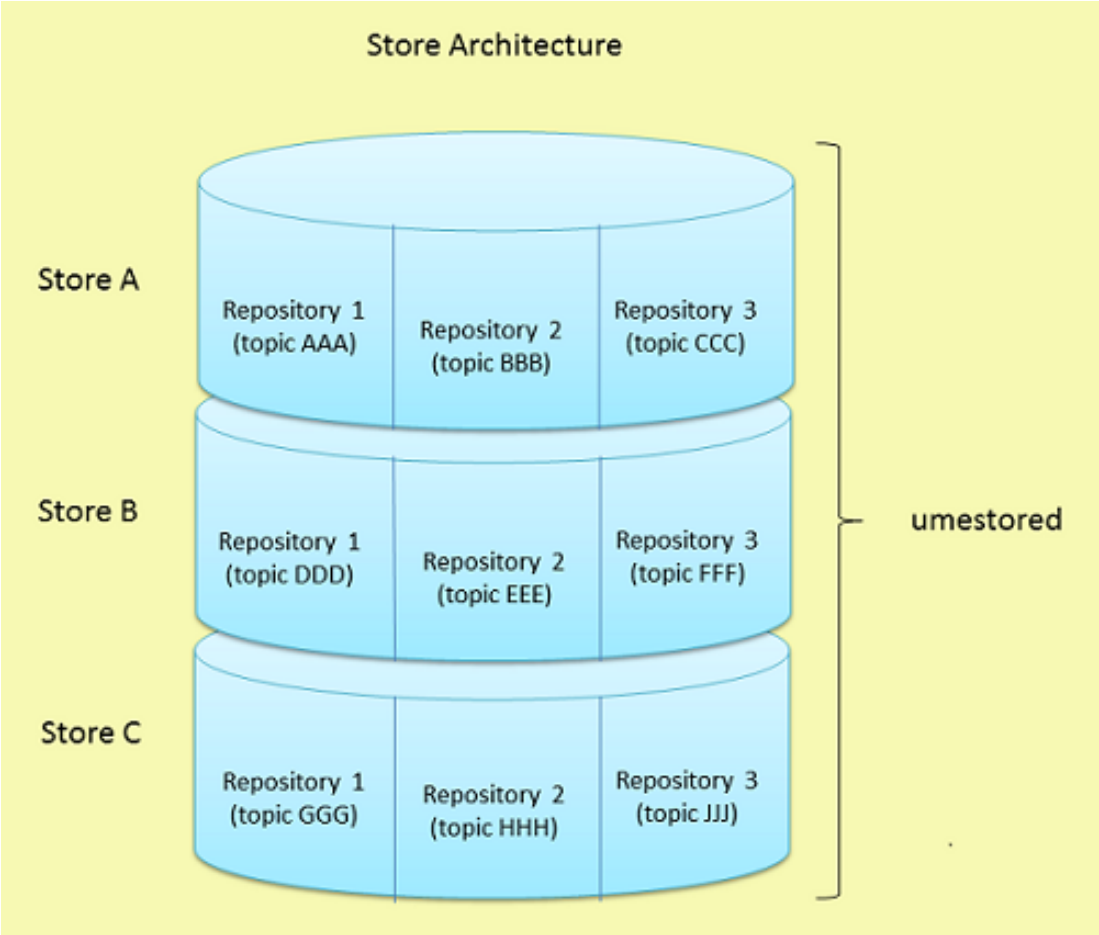
The highlights of this architecture are:

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

Note that the store is not supported on all platforms. For example, while OpenVMS supports persistent clients (source and receiver), you cannot run a store on an OpenVMS system. However, an OpenVMS-based client can interoperate with a store running an any other supported platform.

4.1 Persistent Store Architecture

The umestored daemon runs the 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. UM provides each umestored daemon with a Web Monitor for statistics monitoring. See Store Web Monitor.



4.1.1 Source Repositories

Within a store, you configure repositories for individual topics and each can have their own set 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, UM creates a separate repository for each source. UM 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.

4.1.2 Repository Thresholds and Limits

Repositories are designed as circular buffers. When age or size thresholds are met for a topic, the repository removes or overwrites messages in order to prevent reaching its configured limit, which keeps space available for new messages. UM provides UM configuration options and store configuration options to control threshold and limit behavior.

UM configuration options control source repositories for all the sources sending within the context. The default for these options, listed below, are 0 (zero) which makes the like-name option for the repository in the umestored XML configuration file active.

- **ume_repository_disk_file_size_limit (source)**
- **ume_repository_size_limit (source)**
- **ume_repository_size_threshold (source)**

See **Ultra Messaging Persistence Options**.

Note: The above configuration options' default values can be altered for individual sources and receivers by calling **lbm_src_topic_attr_setopt()** before you allocate the topic.

The umestored configuration options for source/topic repositories explained below can also be used to control threshold and limit behavior. See Topic Element for complete information about the following repository options.

Note

Whether you use the UM configuration options mentioned above or the source repository options explained below to control source repository threshold and limit behavior, remember the values you configure apply to a single source sending to the store. If you use the default repository size limit of 48 MB and you have 1,000 sources sending to the store, UM creates a store with 1,000 source repositories of 48 MB each, which requires a store with approximately 48 GB of memory. And if you use the default disk file size limit of 100 MB and you have 1,000 sources sending to the store, UM creates a store with 1,000 source repositories of 100 MB each, which requires a store with disk storage capacity of approximately 100 GB.

Memory Repository

A memory type source repository has three configuration options that manage its size relative to its capacity.

- repository-age-threshold - This value determines how long the repository retains messages. The repository deletes any message older than this configured value.
- repository-size-threshold - The size in bytes that a repository can reach before it begins to delete the oldest retained messages. If the repository size falls below the threshold, it stops deleting old messages.
- repository-size-limit - The maximum size in bytes for the repository. Once this limit is reached, the repository stops accepting new messages. The age and size thresholds should be set at levels that guarantee the size limit is never met. You should consider how fast the source sends messages, the size of the messages and the reliability of the receivers. For example, more reliable receivers mean less recovery instances, which could mean a younger age threshold.

Disk or Reduced-fd Repositories

A disk or reduced-fd type source repository maintains a memory cache in addition to the actual disk storage. It continually persists messages from the memory cache to the disk, and uses the memory cache for receiver recovery first before performing disk reads to access needed messages. It has four configuration options that manage its size relative to its capacity.

- repository-age-threshold - This value determines how long the disk repository retains messages in its memory cache. The repository deletes any message from memory cache older than this configured value. These messages could have been persisted to disk and may be available for recovery.
- repository-size-threshold - The size in bytes that a repository can reach before it begins to delete the oldest retained messages. These messages could have been persisted to disk and may be available for recovery. If the disk repository memory cache size falls below the threshold, it stops deleting old messages.
- repository-size-limit - The maximum size in bytes for the disk repository's memory cache. Once this limit is reached, the repository stops accepting new messages. The age and size thresholds should be set at levels that guarantee the size limit is never met. You should consider how fast the source sends messages, the size of the messages and the reliability of the receivers. For example, more reliable receivers mean less recovery instances, which could mean a younger age threshold.
- repository-disk-file-size-limit - The maximum disk space (in bytes) for the disk repository. Once this limit is reached, the repository overwrites old messages with new messages. Overwriting old messages is not necessarily a negative situation provided you disk file size is adequate. However, if messages needed for recovery are not in either the memory cache or the disk file, you may need to increase the disk file size to ensure that overwritten messages are no longer needed for receiver recovery.

4.1.3 Tolerance 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 Quorum/Consensus arrangement for fault tolerance. Be aware that the arrangement of stores into Quorum/Consensus groups is a function of the source. I.e. the individual stores of a Quorum/Consensus group are not aware of each other and do not coordinate their activities.

4.1.4 Identifying Persistent Stores

You can identify stores with either a domainID:interface:port, interface:port or a name. Using only interface:port is more feasible in smaller implementations where the smaller number of possible IP addresses is easier to manage. Larger implementations, especially those that span topic resolution domains using UM Routers, are better served with stores identified by a name or domainID:interface:port.

UM automatically resolves and maintains a mapping between a store name and a single topic resolution domain, IP address and port. UM also automatically resolves store names if the store is located across one or more UM Routers in a different topic resolution domain.

The following lists other specifics of store identification.

- Store sends ads at startup and in response to queries from sources.
- If a store receives a context name advertisement that matches its own store name, umestored issues a warning in the store's log.
- Sources using named stores issue an information message to the application every time a resolved context name changes its DomainID:IPaddress:port.

Using a Single Interface and Port

Configure store for a single interface and port.

1. Identify the store with only the interface:port, specified in umestored configuration file.

   ```
   <store name="newyork-1" port="14567" interface="10.29.3.16">
   ```

2. Add the interface:port to **ume_store (source)** so sources can find and register with the store.

   ```
   source ume_store 10.29.3.16:14567
   ```

To run the store on a different machine for any reason, you must change both the umestored XML configuration file and the UM configuration file.

Using a Range of Interfaces

Configure a store with a range of IP addresses.

1. Identify the store with a range of interfaces specified in the umestored configuration file.

   ```
   <store name="newyork-1" port="14567" interface="10.29.3.16/25">`
   ```

2. Add the active interface to **ume_store (source)** so sources can find and register with the store. You can only specify one interface in the configuration file.

   ```
   source ume_store 10.29.3.16:14567
   ```

To run the store on a different machine, you must only change the interface specified in the **ume_store (source)** UM configuration option, provided you use one of the interfaces in the range specified in in the umestored configuration file.

Using a Store (context) Name

Configure a store with a name instead of just IP:port. '`0.0.0.0`' (`INADDR_ANY`) or no value is the default for the store's interface attribute.

1. Identify the store with a context-name option that resolves to the interface and port - or range of interfaces and port - specified in the umestored configuration file:

   ```
   <store name="newyork-1" port="14567" interface="0.0.0.0">
   <ume-attributes>
      <option name="context-name" type="store" value="NEWYORK-1"/>
   </ume-attributes>
   ```

   OR

   ```
   <store name="newyork-1" port="14567" interface="10.29.3.16">
   <ume-attributes>
      <option name="context-name" type="store" value="NEWYORK-1"/>
   </ume-attributes>
   ```

   OR

   ```
   <store name="newyork-1" port="14567" interface="10.29.3.16/25">
   <ume-attributes>
      <option name="context-name" type="store" value="NEWYORK-1"/>
   </ume-attributes>
   ```

2. Add the store's context name to **ume_store_name (source)** so sources can find and register with the store.

   ```
   source ume_store_name NEWYORK-1
   ```

You do not have to make any configuration changes to run NEWYORK-1 on another machine, provided the new interface matches one of those specified in the umestored configuration file. This includes running the store in a different topic resolution domain.

Chapter 5

Operational View

Sources, receivers, and stores 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 persistence events.

Note

If your application is running with the UM configuration option **request_tcp_bind_request_port (context)** set to zero, request port binding is turned off, which also disables persistence.

5.1 General Persistence Operation

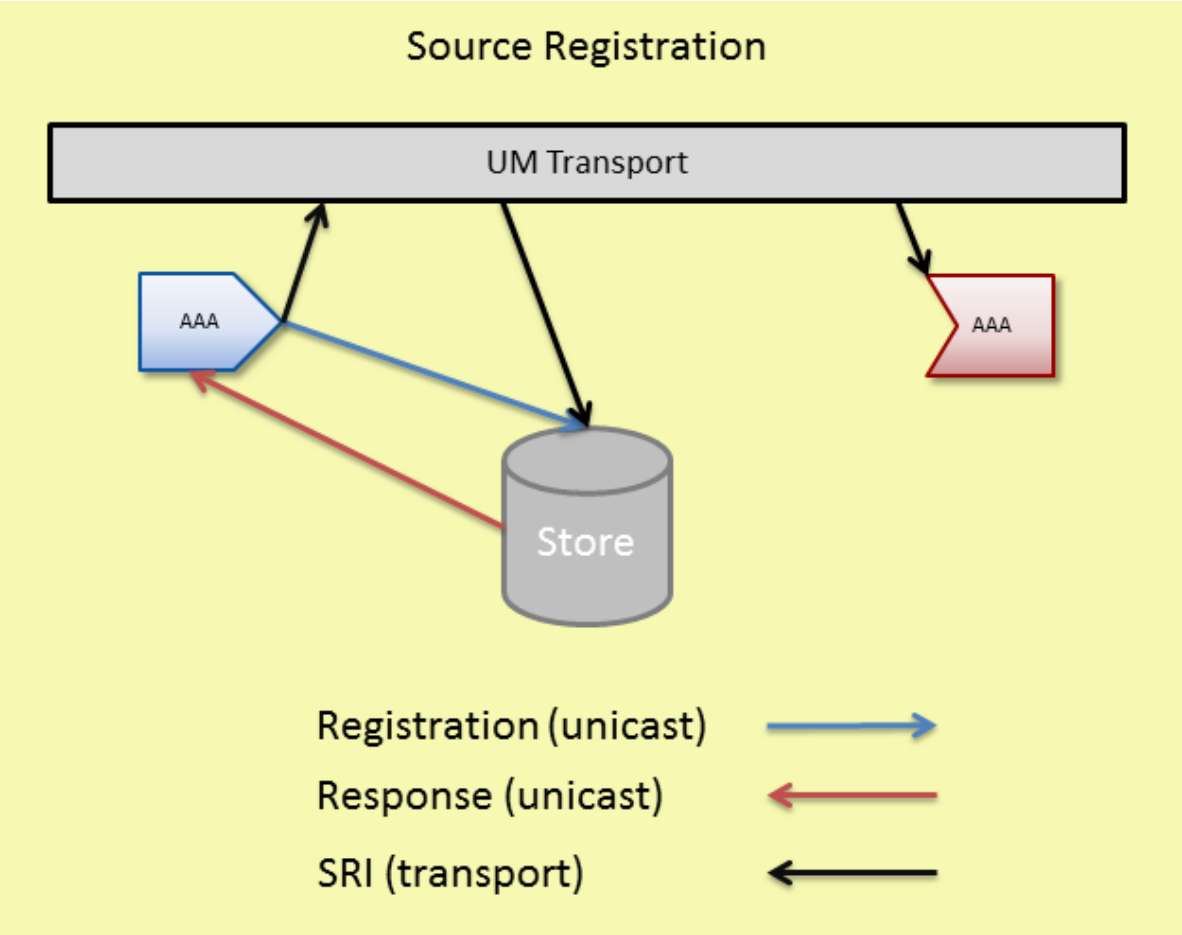
5.1.1 Source Registration

UM sources heavily influence the persistence registration process. Sources send out registration information to enable receivers to register with stores and also monitor store liveness. If stores become unresponsive, or if communication among sources, stores and receivers becomes impaired, the source directs re-registration.

The following outlines the major events in the source registration process with the store:

1. Source advertises topic over topic resolution transport
2. (optional) Source queries for and resolves store name
3. Source registers with store by unicast
4. Source sends SRI over configured transport

The following diagram illustrates network flow during the registration process.



Sources can find the correct store(s) to register with from the values configured for it in **ume_store (source)** or **ume_store_name (source)**. The configuration option **ume_store (source)** contains the IP address, TCP port, registration ID, and group index for the store(s) to be used by the source. The configuration option **ume_store_name (source)** contains the names of the stores to be used by the source. **ume_store_name (source)** requires that the store name is configured with the context-name option in the store's XML configuration file. See Identifying Persistent Stores and the Store Element.

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

After the source successfully registers with all the stores for which it is configured, the source issues a Registration Complete event and sends a Source Registration Information (SRI) record over the configured UM transport session.

For multiple stores, the source determines when to issue a Registration Complete event based on the settings for the **ume_retention_intragroup_stability_behavior (source)** and **ume_retention_intergroup_stability_behavior (source)** options.

The source sends the SRI at the rate set by **ume_sri_inter_sri_interval (source)** until it reaches the maximum number of SRIs set by **ume_sri_max_number_of_sri_per_update (source)**.

Note

Persistence users are advised to follow the recommendations in **Preventing Store Registration Hangs**.

5.1.2 Source Registration Information (SRI)

An SRI is a control message sent over the UM transport by a source that contains store information that a receiver needs to register with the store.

An SRI contains the following store information.

- Domain ID
- IP address
- TCP port
- store index for all the stores with which the source registered
- group index for all the stores with which the source registered
- the source's Registration ID
- SRI overall version number and a separate version number for each store

The SRI contains one overall version number and a separate version number for each store. If stores become unresponsive and the source must re-register when the store returns, the source increases the SRI version number and the version numbers for the stores it re-registered with. The highest SRI version number indicates the most current registration information. If a receiver gets an SRI with a higher version number than the version number it has, the receiver examines the individual store version numbers and re-registers with the those stores that have higher individual version numbers.

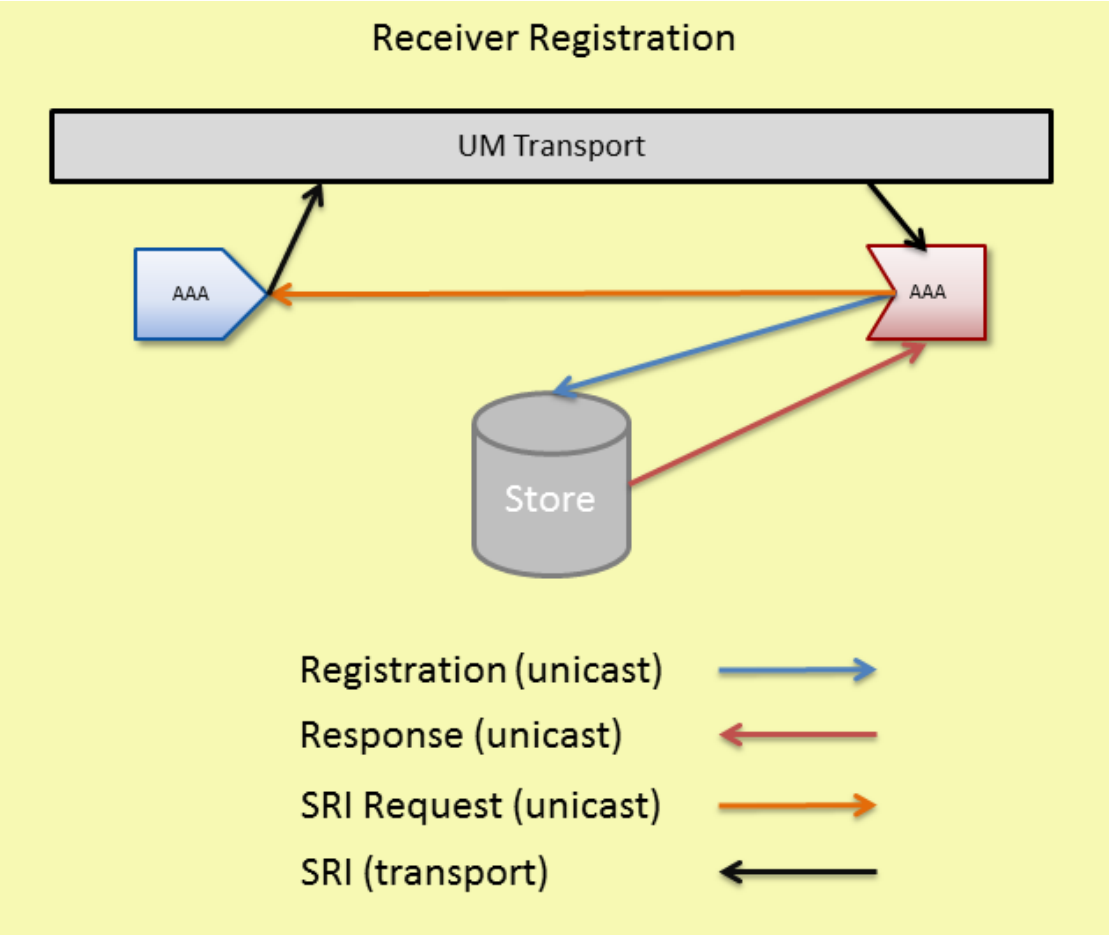
5.2 Receiver Registration

Receivers register with a store or stores after receiving a SRI packet from the source sending on the receiver's topic.

Receiver must receive an SRI before they can register with the store or stores. The following lists the major events in the receiver registration process.

1. Receiver resolves topic over topic resolution transport.
2. If source is not sending SRIs, receiver sends SRI request by unicast.
3. Receiver receives SRI over its transport.
4. Receiver registers with store(s) by unicast.

The following diagram illustrates network flow during the registration process.



5.2.1 Receiver Registration Process

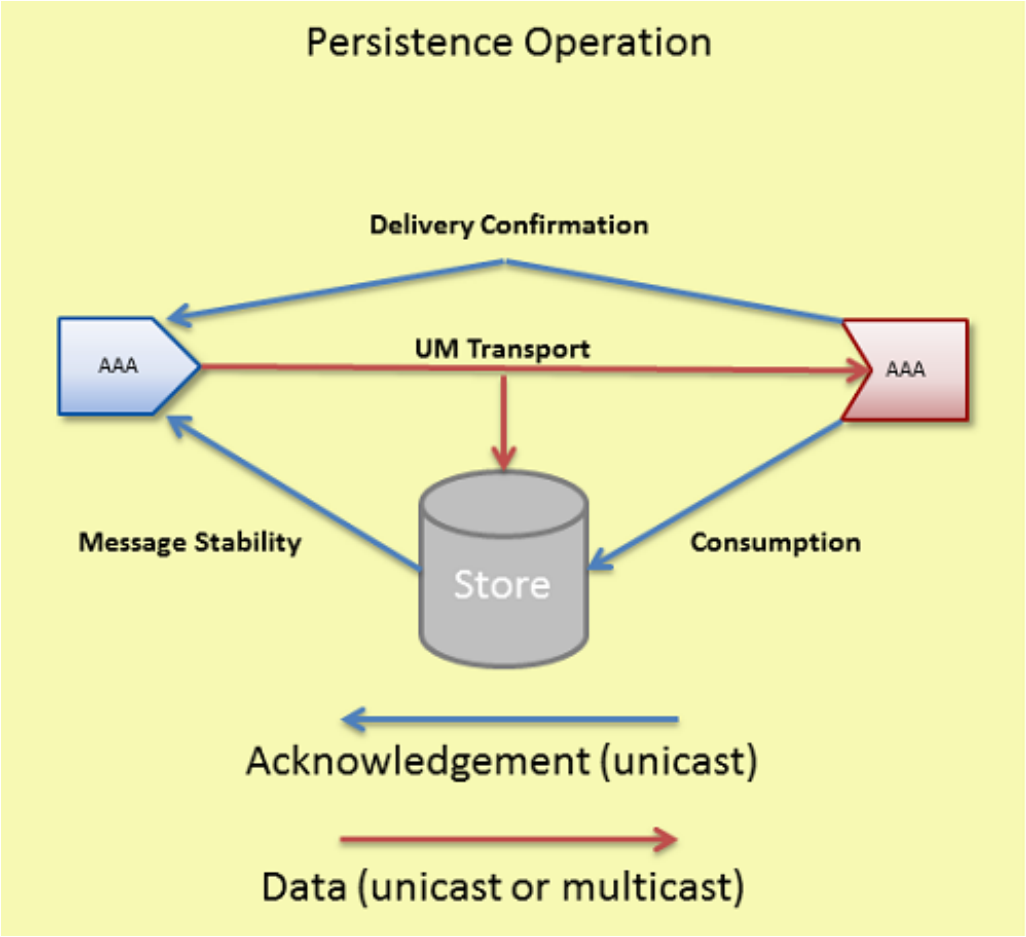
Any receivers who have resolved their topic and joined the transport session when the source sends out SRIs can register with the store. Any receivers joining the transport session when the source is not sending SRIs can request an SRI from the source if they find that the persistence flag is set in the source's TIR during topic resolution. The source responds with a SRI record.

Receivers unicast registrations to the store. The store unicasts responses back to the receivers. Stores use RegIDs to identify sources and receivers. After registration, receivers may handle recovery and send acknowledgements.

Note: If a persistent receiver's initial registration fails, it does not become an Ultra Messaging receiver.

5.2.2 Persistence Normal Operation

The following diagram illustrates the normal operation of data reception and acknowledgement and also shows how UM attains Parallel Persistence. The source sends message data to receivers and stores in parallel.



During normal persistence 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).

5.2.3 Persistence Flight Size

UM supports a flight size mechanism that tracks messages in flight from a persistent source and responds when a send would exceed the configured flight size (**ume_flight_size (source)** and/or **ume_flight_size_bytes (source)**). You can configure **ume_flight_size_behavior (source)** to either:

- block any sends that would exceed the flight size or,

- allow the sends while notifying your application.

UM considers a sent message in flight until the following two conditions are met:

- The source receives the configured number of stability acknowledgements from the store(s).
- The source has received the configured number of delivery confirmation notifications. (See **ume_retention_unique_confirmations (source)**.)

If configuring both **ume_flight_size (source)** and **ume_flight_size_behavior (source)**, UM uses the smaller of the two flight sizes on a per send basis.

ume_flight_size (source)	ume_flight_size_bytes (source)	Result
Exceeded	Exceeded	**ume_flight_size_behavior (source)** executes
Exceeded	Not Exceeded	**ume_flight_size_behavior (source)** executes
Not Exceeded	Exceeded	**ume_flight_size_behavior (source)** executes
Not Exceeded	Not Exceeded	No flight size sending restriction

When using stores in a Quorum/Consensus configuration, intragroup and intergroup stability settings affect whether UM 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, UM considers a given message in-flight. In addition, if you set **ume_retention_unique_confirmations (source)** 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.

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.

Notification of Message Sends That Exceed the Flight Size

Alternatively, **ume_flight_size_behavior (source)** 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.

5.2.4 Receiver Recovery

Normal loss retransmission over the UM transport operates identically in persistence as it does in streaming, according to the transport protocol. Stores do not participate in this transport-level loss retransmissions.

Persistent stores become involved in message recovery in circumstances where the transport protocol is not able to recover. For example, if an application exits (either intentionally or by failure) and then restarts some time later, the transport is not able to recover messages that were sent during the application's down time. When the receiver restarts and re-registers, the receiver discovers the lowest message sequence number it did not receive, and subsequently requests retransmissions of all messages not received, starting from this low sequence number.

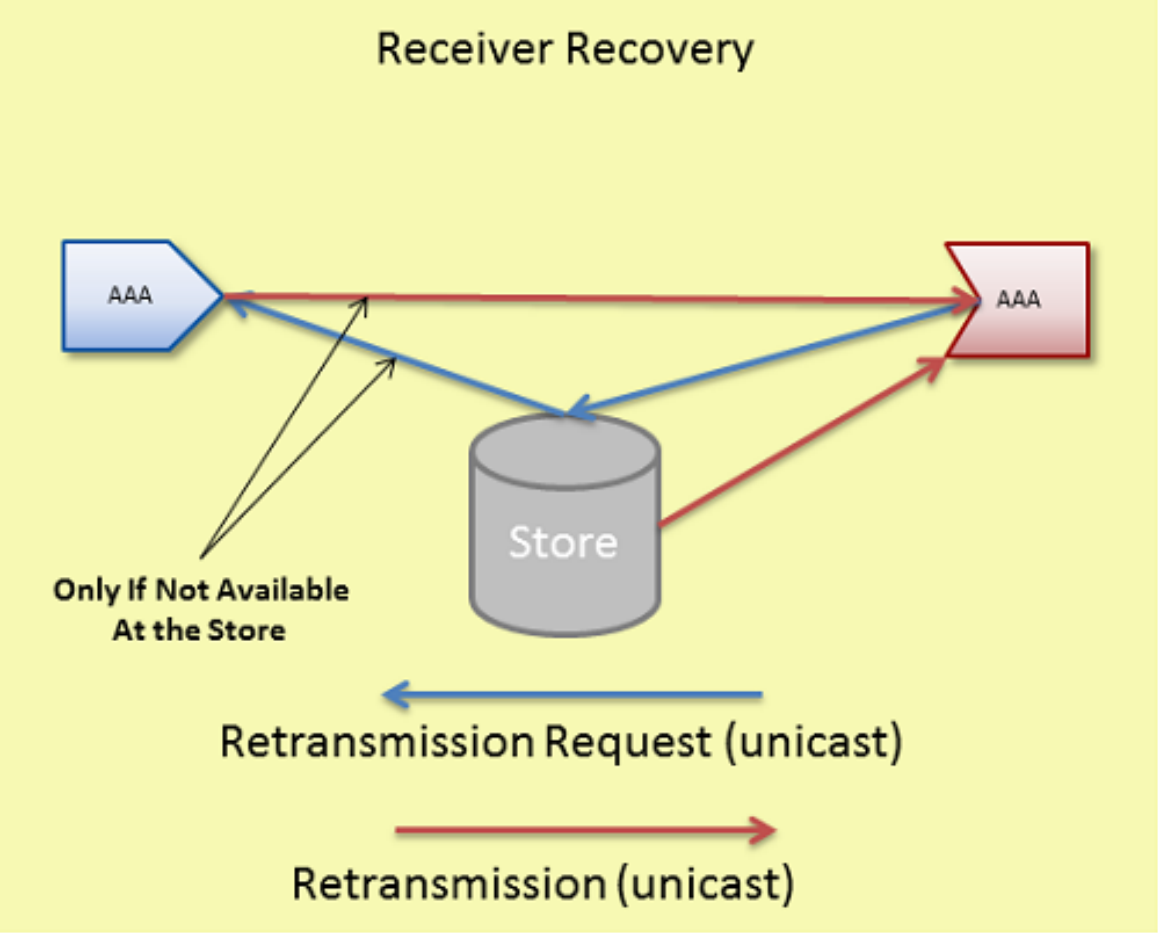
For more on this process see, Persistent Receiver Recovery.

Another circumstance in which the store becomes involved in message recovery is if the transport protocol tries but is unable to recover lost messages. In this case, Off Transport Recovery (OTR) is used. Note that OTR is available in streaming, and is serviced by the source's retention buffer. But for persistent sources, the store services OTR.

See **Off-Transport Recovery (OTR)** for more information.

For more reliable persistence operation, Informatica recommends enabling OTR, especially when using UM Routers.

The following diagram illustrates receiver 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, it unicasts the retransmission request to the source. If the source has the message, it unicasts the retransmission directly to the receiver. See also Message Loss Recovery.

UM store 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 indicate retransmission activity (see Store Web Monitor):

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

5.3 Receiver-paced Persistence Operations

The Receiver-paced Persistence mode of operation is primarily intended to prevent message loss to critical receivers, even if loss prevention requires blocking sources from sending. To achieve this, message retention in the store is different from Source-paced persistence:

- In Source-paced Persistence (SPP), messages are retained in the store until the space is needed for new messages. I.e. the message repository is a circular buffer which will overwrite when it "wraps". If a slow or stopped receiver falls behind the source by more than the size of the store's repository, that receiver will experience unrecoverable loss.
- In Receiver-paced Persistence (RPP), messages are retained only for as long as registered receivers need them to be retained in order to ensure recoverability of unacknowledged messages. When all necessary receivers have acknowledged a message, that message is removed from the store's repository. If critical receivers are unable to acknowledge messages and the repository has reached its configured capacity, the source is blocked from sending additional messages. Blocking the source prevents sending of messages that would otherwise overwrite unacknowledged messages.

Source pacing is typically chosen for applications where outgoing messages are generated by external events or processes that cannot be slowed down or stopped (e.g. market data). Receiver pacing is typically chosen for applications which are able to slow down or even halt the generation of messages (e.g. a user interface which can inhibit user entry).

RPP is enabled with UM configuration options. No special API calls are needed.

RPP differentiates between two types of receivers:

- Blocking: A blocking receiver will block the source if additional messages would overwrite retained messages not yet acknowledged by that receiver.
- Non-blocking: A non-blocking receiver will not block the source; the source will be allowed to overwrite retained messages not yet acknowledged by the non-blocking receiver. Thus a non-blocking receiver will experience unrecoverable message loss if it falls behind the source by more than the configured size of the store's repository. (Note that this is the same behavior of source-paced persistence.)

Each receiver indicates its desired blocking behavior with the **ume_receiver_paced_persistence (receiver)** configuration option. Both blocking and non-blocking receivers may register with the same store and subscribe to the same source.

Here are important points when using RPP:

- The repository must be configured to allow RPP, and sources and receivers must be configured to request RPP behavior during registration. Assuming the store is configured to allow RPP, the source determines the pacing behavior (receiver v.s. source) when it registers. If a receiver requests a different behavior, its registration will fail.
- The store tracks the number of registered blocking and non-blocking receivers for each message sent by the source. A message is normally retained in the store repository until that number of receivers have acknowledged consumption. Once all receivers acknowledge consumption of a message, that message is removed from the repository.
- Sources can modify specific repository configuration options that pertain to RPP.
- Due to RPP's message retention policies, late joining RPP receivers cannot recover previously sent messages.
- With RPP, sources are required to configure their flight size in bytes, in addition to message count. (With SPP, only message count flight size is required.) The value set for the source's **ume_flight_size_bytes (source)** configuration option is checked against a maximum allowed value specified in the store's XML configuration file.

- With RPP, if the store's repository is full with unacknowledged messages by blocking receivers, the store will block the source by withholding stability acknowledgements, resulting in flight size blockage. See Persistence Flight Size. (With SPP, once the repository is full, it will simply start overwriting the oldest messages with new messages from the source.)

In addition, a disk write delay interval for the repository, improves performance by preventing unnecessary disk activity.

RPP introduces the capability of a source application to set the following operational options on the store:

- repository-size-threshold
- repository-size-limit
- repository-disk-file-size-limit
- repository-disk-write-delay

With SPP, those parameters are set only by the store's XML configuration file alone. With RPP, the source's configuration can optionally request a different value for those operating parameters, with the store's configured value being used as a maximum allowed threshold.

5.3.1 RPP Registration

A source configures its desired pacing behavior (source paced v.s. receiver paced) with **ume_receiver_paced_persistence (source)** and **ume_receiver_paced_persistence (receiver)**. If set to 1, it becomes an RPP source. Assuming the store is configured to allow RPP, when an RPP source registers with the store, the store's repository for that source becomes an RPP repository. The receiver configures its desired pacing behavior with **ume_receiver_paced_persistence (receiver)**, where 0 is source-paced and 1 or 2 are receiver-paced. The receiver's pacing must match that of the source and store, otherwise the receiver's registration will fail. In addition, the choice of 1 or 2 determines the receiver's desired blocking behavior (1=blocking, 2=non-blocking).

Note that although the configured pacing behavior must match between source and receiver, that does not mean that the numerical setting of the **ume_receiver_paced_persistence (source)** and **ume_receiver_paced_persistence (receiver)** options must be equal. If the source is 0 (source paced), then the receiver must also be 0. However, if the source is 1 (receiver paced), then the receiver must be either 1 or 2, depending on the receiver's desired blocking behavior.

As with Source-paced Persistence, RPP sources send Source Registration Information (SRI) packets to RPP receivers over the configured UM transport. RPP Receivers must wait for this information before they can initiate registration requests to the store. See Source Registration and Receiver Registration for more information.

A source registration request includes the following:

- Designation of an RPP topic
- 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 the same configuration options as were initially requested, or the store will reject the request.

A receiver registration request includes its designation as a RPP receiver.

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

Late Registering Receiver

A late joining receiver that registers after the first RPP topic message has been sent cannot recover any messages sent prior to its initial registration. It is the user's responsibility to synchronize a receiver's initial registration with

the start of message transmission. This restriction does not apply to an RPP receiver that initially registered at an earlier time and is now re-registering, as after a failure and restart. In that case, messages that were sent after the receiver's initial registration will be retained by the store for recovery by the receiver.

Early Exiting Receiver

Each registered receiver has associated with it an activity timeout and a state lifetime. During normal operation, the store monitors the operation of a registered receiver. If the store hears nothing from a receiver for the duration of the activity timeout, the store assumes that the receiver has halted operation. Messages will be retained by the store according to the receiver's configured blocking behavior. This gives the receiver time to restart and re-register. If an inactive receiver re-registers before the state lifetime expires, the receiver will be able to recover all messages that it missed.

However, if a receiver remains halted for the duration of the state lifetime, the store will delete the receiver state information. If the repository is retaining messages for this receiver, those messages will be implicitly acknowledged on behalf of the expired receiver, making them eligible for deletion if no other receivers' acknowledgements are pending. If the source is blocked waiting for this receiver, the store will unblock the source. Finally, if the halted receiver re-register after its state lifetime has expired, the store will treat it as an initial registration, and the messages it missed will not be available.

UM Version RPP Compatibility Matrix

The following table indicates the result of registration requests across UM 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

Where:

- 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.

5.3.2 RPP Normal Operation

At a high level, the normal sequence of operations for RPP is the same as it is for SPP:

1. Sources transmit messages to receivers and stores at the same time over UM transports. Sources also track stability acknowledgements from the store. A source is allowed to send messages ahead of stability acknowledgements up to the configured flight size. If the flight size of unstabilized messages is reached, the source is blocked from sending more messages pending stability acknowledgements from the store.

2. Receivers acknowledge consumption of received messages back to stores, and optionally to the sources.

3. Stores retain messages as appropriate, send stability acknowledgements to the sources for messages, and tracks receiver consumption acknowledgements.

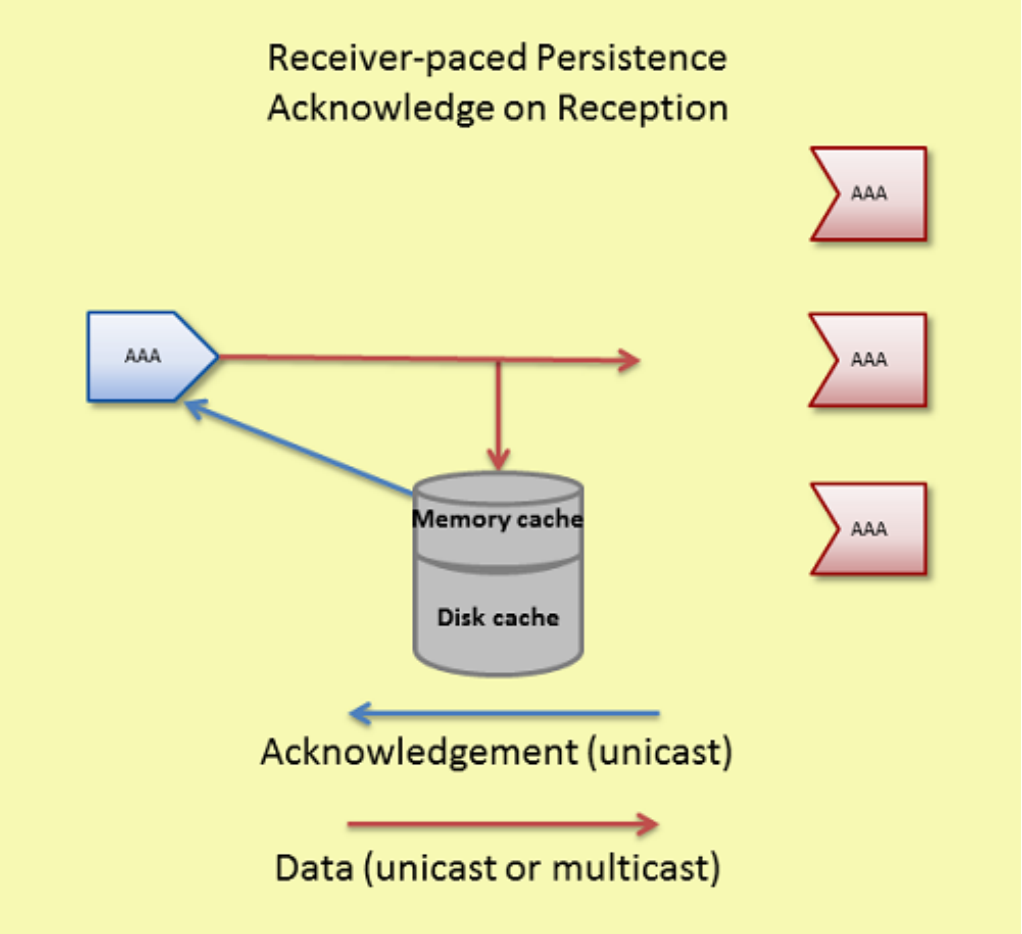
One important way that RPP differs from SPP is in the sending of stability acknowledgements. With SPP, the store normally waits to send a stability acknowledgement until a message is "stable" on the configured storage medium, either disk or memory. With RPP, the sending of stability acknowledgements is affected by receiver consumption acknowledgements in two ways:

- If a message is acknowledged by all registered receivers before the message is written to disk, then there is no need to retain the message at all. The message is deleted and a stability acknowledgement is sent to the source.

- If the repository reaches its capacity limit and there are blocking receivers which have not acknowledged the messages, the store stops sending stability acknowledgements. It is the lack of stability acknowledgements, combined with the configured flight size, which causes the source to block. (To be precise, the store stops sending stability when there is exactly one flight size worth of room available in the repository.)

The following also affect stability acknowledgements:

- Acknowledge on Reception - If the source is configured for **ume_repository_ack_on_reception (source)** and the store is configured for repository-allow-ack-on-reception, the store sends a stability acknowledgement to the source immediately upon reception of a message, even before any receiver acknowledgements are received, and before the message is written to disk. This setting can increase system throughput for some use cases, but also increases the risk of message loss in the event of a store failure.



- 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.

For memory store repositories, the options **ume_repository_ack_on_reception (source)** and repository-disk-write-delay have no effect.

5.3.3 RPP Message Recovery

The normal way that RPP receivers recover messages is when they re-register within the state lifetime after a failure. However, just as with SPP, there is the possibility that the transport session of the source is unable to successfully deliver all messages to the receiver. In the event of unrecoverable loss at the transport session, the Off Transport Recovery (OTR) method is also active for RPP receivers. OTR does not require the receiver to restart to recover messages from the store. See the **Off-Transport Recovery (OTR)** for more information.

5.3.4 RPP Deregistration

You can deregister either sources or receivers using deregistration APIs, (**lbm_src_ume_deregister()**, **lbm_rcv_ume_deregister()**, and **lbm_wrcv_ume_deregister()**). UM deletes the state of deregistered objects. If you deregister an RPP receiver, UM automatically decrements the number of receiver acknowledgements required to maintain RPP behavior. The store issues Deregistration Successful events for every source or receiver that deregisters. Note that after deregistering a source or receiver, the object will still exist, but is no longer participating in persistence. An attempt to send to a deregistered source will return an error. A deregistered receiver will continue to deliver messages on the topic, but since it is no longer participating in persistence, it will be unable to acknowledge those messages. If the application wants to re-join persistence, it must delete the source or receiver and re-create it, allowing it to re-register. See Persistence Events.

Users should be cautious using the deregistration APIs, especially for sources. Source deregistration will immediately delete from the store any messages from that source which might be retained due to lack of receiver acknowledgement. This deletion will render the receivers unable to recover those messages.

5.3.5 Implementing RPP

Follow the procedure below to configure Receiver-paced Persistence:

1. Set **ume_receiver_paced_persistence (source)** and **ume_receiver_paced_persistence (receiver)** in the UM configurations. If only certain sources or receivers in a context are RPP, use lbm_∗setopt() in the source or receiver application or use UM XML configuration files.
2. Set repository-allow-receiver-paced-persistence = 1 for the repository in the umestored XML configuration file.
3. Coordinate **ume_flight_size_bytes (source)** between the repository and the source. Set the maximum allowable flight size with the repository option, source-flight-size-bytes-maximum. Sources can reconfigure its flight size bytes to a value less than or equal to the maximum.
4. Optional: coordinate the **ume_repository_ack_on_reception (source)** 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. 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 set **ume_write_delay (source)** to a value less than or equal to repository-disk-write-delay.

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 (source)**
   - **ume_repository_size_limit (source)**
   - **ume_repository_disk_file_size_limit (source)**

5.3.6 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.

UM Configuration File for RPP

The following example UM configuration file will work for applications which have sources and/or receivers that must be persisted using RPP. This configuration file is written assuming that the store is configured as shown in the next section.

- The source configures **ume_flight_size_bytes (source)** to 1,000,000 bytes. For this to work, the repository must set source-flight-size-bytes-maximum to a value greater than or equal to 1,000,000.
- The source uses **ume_write_delay (source)** to override the repository's repository-disk-write-delay setting to 1000 ms (1 second). Note that for this to work, the repository must set repository-disk-write-delay to a value greater than or equal to 1000 ms.
- To remove clutter from the example, the transport type is allowed to default to TCP. Many persistence users prefer LBT-RM to more quickly and efficiently distribute messages to stores and receivers.

```
##Sample UM Configuration File
# Default to TCP transport
# Multicast Resolver Network Options
context resolver_multicast_address 225.8.17.29
context resolver_multicast_interface 10.29.3.0/24

## Persistence Options ###
source ume_store_name store0rpp
source ume_store_name store1rpp
source ume_store_name store2rpp
source ume_session_id 535353
source ume_store_behavior qc
source ume_flight_size 500
# RPP-oriented configs.
#   If this app creates receivers, have them request RPP mode.
receiver ume_receiver_paced_persistence 1
#   If this app creates sources, have them request RPP mode.
source ume_receiver_paced_persistence 1
source ume_flight_size_bytes 1000000
#   The following parameters override store configurations.
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
```

umestored Configuration File

In the following example store configuration file, RPP options appear in the section for the topic pattern, ABC∗. This configuration file is written assuming client applications (sources and receivers) use UM configuration files similar to that shown in the preceding section.

There are actually three stores configured in Q/C. The other two's configurations should differ appropriately. For example, change each instance of "store0" to "store1" and "store2" respectively.

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <log>/configs/stores/umestore0/umestored.log</log>
    <pidfile>/configs/stores/umestore0/umestored.pid</pidfile>
    <lbm-license-file>/bin/umq_exp_license.txt</lbm-license-file>
    <lbm-config>/configs/lbm_store0.cfg</lbm-config>
    <web-monitor>*:15404</web-monitor>
  </daemon>
  <stores>
    <store name="rpp-ump-test-store0" 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="context-name" value="store0rpp"/>
      </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="source-flight-size-bytes-maximum"
                 value="4194304"/>
             <option type="store" name="repository-allow-ack-on-reception"
                 value="1"/>
             <option type="store" name="repository-disk-write-delay" value="1000"/>
          </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
</ume-store>
```

5.3.7 RPP Cross Feature Functionality

UM Feature	Supported	Notes
Store Proxy Sources	Yes	

UM Feature	Supported	Notes
UM Router	Yes	
UM Transports	Yes	
Multi-Transport Threads	No	The Multi-Transport Threads does not support persistence.
Off-Transport Recovery	Yes	
Late Join	No	A receiver cannot recover messages sent prior to that receiver's initial registration.
HF	Yes	
HFX	Yes	
Wildcard Receivers	Yes	
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 - Quorum Consensus	Yes	
UM SNMP Agent	Yes	
Ultra Messaging Manager	Yes	
Ultra Messaging Cache	Yes	
Ultra Messaging Desktop Services	No	

5.4 Persistence Events

The Ultra Messaging API provides a number of events, callbacks, messages, functions, and settings. The API reference (C API, Java API or .NET API) 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 application callbacks called directly from UM 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 UM 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, Java API or .NET API).

5.4.1 Persistence 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
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.
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 message meets Intergroup and/or Intragroup stability requirements. Also includes the store information.
Store Message Not Stable	Source Event	Delivered once a message's **ume_message_stability_lifetime (source)** has expired. The source no longer retransmits the message to the store.
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. If a majority of a source's configured stores are unresponsive, the application will not be allowed to send messages.
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 (source)**) has been exceeded or the message's **ume_message_stability_lifetime (source)** has expired. Event contains the message sequence number.

Event Name	Type	Description
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).
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 UM receives a successful deregistration event from all stores.

5.4.2 Persistence 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.
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.

Event Name	Type	Description
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 UM receives a successful deregistration event from all stores.
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.
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.
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 UM 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 UM 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.

5.4.3 Persistence Context Events

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

Event Name	Type	Description
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 (UND↩ ER).

Chapter 6

Enabling Persistence

The following table lists all source files used in this section. The files can be found in the /doc/example directory. You can also access these file via the Sample Source Code tab in the left panel, under C Example Source Code.

Filename	Content
ume-example-src.c	Source Application
ume-example-rcv.c	Receiver Application
ume-example-src-2.c	Source Application 2
ume-example-rcv-2.c	Receiver Application 2
ume-example-src-3.c	Source Application 3
ume-example-rcv-3.c	Receiver Application 3
ume-example-config.xml	Persistent Store Configuration File

6.1 Starting Configuration

We begin with the minimal source and receiver used by the QuickStart Guide. To more easily demonstrate the persistence 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 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 UM provides up front.

Example files for our exercise are:

Filename	Content
ume-example-src.↩ c	Source Application
ume-example-rcv.↩ c	Receiver Application

6.2 Adding the Store to a Source

The fundamental component of a persistence solution is the persistent store. To use a store, a source needs to be configured to use one by setting **ume_store (source)** 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 persistent store for the source to the store running at 127.0.0.1 on port 14567.

Example files for our exercise are:

Filename	Content
ume-example-src-2.c	Source Application 2
ume-example-rcv-2.c	Receiver Application 2
ume-example-config.xml	Persistent Store Configuration File

After adding the ume-store specification to the source, perform the following steps (assumes a Unix command prompt):

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 UM and merely means Late Join was not set and that UM 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 UM 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.

6.3 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 persistence, these identifiers are called Registration IDs or RegIDs. UM 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, UM 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 a callback function which will return the RegID value 1100 to UM. The example names the callback `app_rcv_regid_callback()`.

2. Inform the UM configuration for the receiver to use this callback function. That is accomplished by setting the ume_registration_extended_function() 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.

6.4 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. The sources and receivers are primarily responsible for managing the RegIDs.

The following two sections explain the changes needed for the source and receiver, which become fairly easy due to the events that UM delivers to the application during persistence operation.

6.5 Enabling Persistence in the Source

With the above mentioned behaviors in mind, let's turn to looking at how they may be implemented with persistence, 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 (source)** 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 UM to establish the beginning message number because UM will use the next sequence number. For this simple example, we can make the assumption that each message is one sequence number for UM and that UM starts with sequence number 0. Thus the application can set the message number it begins resending with the value of the UM sequence number + 1. These changes yield the source code in `ume-example-src-3.c`

6.5.1 Smart Sources and Persistence

When using the **Smart Sources** feature to send persistent messages, there are a few restrictions:

- No support for source-side delivery confirmation. Neither of the forms described in Delivery Confirmation Concept are allowed.
- No support for Receiver Liveness Detection.
- Application stability notification is only supported per-message. See **ume_message_stability_notification (source)**.
- The following configuration options have limited or no support with Smart Sources:
  - **ume_confirmed_delivery_notification (source)**
  - **ume_retention_unique_confirmations (source)**
  - **ume_sri_flush_sri_request_response (source)**
  - **ume_sri_request_response_latency (source)**
  - **ume_message_stability_notification (source)**
  - **retransmit_retention_size_threshold (source)**
  - **ume_retention_size_threshold (source)**
  - **retransmit_retention_size_limit (source)**
  - **ume_retention_size_limit (source)**
  - **retransmit_retention_age_threshold (source)**

6.6 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. UM 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 UM sequence number 19), end the application and delete the saved RegID file.

RegIDs in UM 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-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.

Chapter 7

Demonstrating Persistence

The following files are used in this section:

Filename	Content
ume-example-src-3.c	Source Application 3
ume-example-rcv-3.c	Receiver Application 3
ume-example-config.xml	Persistent Store Configuration File

Perform the following tasks first:

1. Build ume-example-rcv-3.c and ume-example-src-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 UM store. See Store Web Monitor.

7.1 Running Persistent 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:
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 named '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.

7.2 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. Let it run for a few seconds so the receiver gets a few messages.

```
$ ume-example-src-3.exe
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.

7.3 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.

7.4 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 Designing Persistent 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`, `umesrc.java`, or `umesrc.cs` for examples of this behavior.

Chapter 8

Designing Persistence Applications

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 UM's flexible methods of persistence to add 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 persistence features:

- Registration Identifiers
- Designing Persistent Sources
- Designing Persistent Receivers
- Designing Persistent Stores

8.1 Registration Identifiers

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

- **Recommended**: use Session IDs to enable the Store to both assign and manage RegIDs. See Managing RegIDs with Session IDs. Note: while the use of Session IDs is recommended, an understanding of the underlying registration IDs is often helpful to understanding persistence.
- 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 Stores to assign RegIDs and then save the assigned RegIDs for subsequent reuse. See Save Assigned RegIDs.

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.

8.1.1 Use Static RegIDs

For very small deployments, the simplest method uses static RegIDs for individual applications. This method requires every persistent source connecting to a given store have a unique RegID from every other persistent source attaching to the same store. This includes publishing applications that have multiple persistent topics; each topic's source object must have a unique RegID. (The use of session IDs greatly simplifies the management of these RegIDs.)

The following source code examples assign a static RegID to a source by adding the RegID, 1000, to the **ume_store (source)** 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);
}
```

8.1.2 Save Assigned RegIDs

When using RegIDs, your application can request that the store assign it a new and unique RegID when it registers for the first time. That RegID is made available to the application, which can then save it to local storage. Thus, the next time the application starts (or restarts) and wants to use the same registration, it reads the value written to local storage. This method of managing RegIDs is not common. For example, what if the application needs to be restarted on a different server due to hardware failure? If it cannot re-register with its earlier RegID, it will not be able to recover only those messages it had not yet acknowledged. (The use of Session IDs simplifies this greatly by essentially saving the registration IDs for you on the store itself.)

The following minimal source code example saves the RegID assigned to a source to a file. See also ume-example-src-3.c

C API

```
/
* Callback invoked by UM for source events. */
int app_src_callback(lbm_src_t *src, int event, void *eventd, void *clientd)
{
  ...
  switch (event) {
  ...
  case LBM_SRC_EVENT_UME_REGISTRATION_SUCCESS_EX:/
    * Get the registration information. */
    lbm_src_event_ume_registration_ex_t *reginfo =
        (lbm_src_event_ume_registration_ex_t *)eventd;/

    * Might want to do the following conditionally only if we are requesting a new
        RegID. */
    FILE *fp = fopen("UME-example-src-RegID", "w"); / * Error checking omitted for
        clarity. */
    fprintf(fp, "%s:%u", reginfo->store, reginfo->registration_id);
    fclose(fp);
  ...
  } / * switch */
  ...
} / * app_src_callback */

...

err = lbm_src_create(&src, ctx, topic, app_src_callback, ...); / * Error checking
    omitted. */
```

8.1.3 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 UM stores to correctly identify all the sources and receivers for a particular application.

In practice, a Session ID is often thought of as an application identifier, although it is more accurately thought of as a context identifier. (For applications that only have a single context with persistent sources and/or receivers, the two are effectively the same.) However, be aware that many application systems run multiple instances of a given program, perhaps for horizontal scaling. Each instance needs its own Session ID.

It is also possible for a single context to host multiple Session IDs, although this is rarely done. The UM configuration options **ume_session_id (source)** and **ume_session_id (receiver)** can be used to arrange individual source and/or receiver objects into registration groupings. However, it is more common to use the option **ume_session_id (context)** to group all sources and receivers created within a context into a single session ID. (If both a context and a source or receiver option is specified, the source or receiver option will override the context option.)

How Stores Associate Session IDs and RegIDs

Session IDs do not replace the use of RegIDs by UM 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.

All of the above steps happen within UM itself without any intervention by the application. However, the application does have access to the underlying registration ID, if it desires it.

8.2 Designing Persistent Sources

The major concerns of sources revolve around RegID management and message retention.

8.2.1 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. The UM library 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 UM, as opposed to your application, to accomplish the same RegID management. See Managing RegIDs with Session IDs Managing RegIDs with Session IDs.

8.2.2 Sources Must Be Able to Resume Sending

A source sends messages unless UM 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;
}
```

8.2.3 Source Message Retention and Release

UM allows streaming of messages from a source without regard to message stability at a store, which is one reason for UM'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.

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).

- Message must have been confirmed as delivered by a configured number of receivers (**ume_retention_↩ unique_confirmations (source)**).
- The aggregate amount of buffered messages exceeds **retransmit_retention_size_threshold (source)** bytes in payload and headers.

Some things to note:

- If **retransmit_retention_size_threshold (source)** is not met, no messages will be released regardless of stability.
- If the source registered with a "no-cache" store (See Persistent Store Concept) or **ume_message_stability↩ _notification (source)** is turned off, **ume_retention_unique_confirmations (source)** is the only way to allow the source to release messages before retention size options come into play.
- With a quorum/consensus store configuration, when a quorum of stores report stability for a message, remaining stores may or may not send additional stability acks for that message.

Note

Smart Sources simplify matters somewhat by pre-allocating retention buffers. They are not dynamically allocated or deallocated during operation. See Smart Sources and Persistence for more information.

8.2.4 Forced Reclaims

If the aggregate amount of buffered messages exceeds **retransmit_retention_size_limit (source)** bytes in payload and headers, then UM forcibly releases the oldest retained message even if it does not meet one or more of the conditions stated in Source Message Retention and Release. This condition should be avoided and Informatica suggests increasing the **retransmit_retention_size_limit (source)**.

A second condition that produces a forced reclaim is when a message remains unstabilized when the **ume_↩ message_stability_lifetime (source)** expires.

Whenever UM performs a Forced Reclaim, it notifies the application in the following ways:

- The source event callback's RECLAIMED_EX event (see Persistence Source Events) includes a "FORC↩ ED" flag on the event. (UM uses the same RECLAIMED_EX event, without the FORCED flag, for normal reclaims.)
- Through the separate forced reclaim callback, if registered. You set this separate forced reclaim callback with the **ume_force_reclaim_function (source)** configuration option.

Note

UM retains the separate callback for backwards compatibility purposes and may be deprecated in future releases. The source event FORCED flag is the recommended method of tracking forced reclaims.

The following sample code, from umesrc.c, implements the extended reclaim source event with the 'Forced' flag set if the reclamation is a forced reclaim.

C API

```
case LBM_SRC_EVENT_UME_MESSAGE_RECLAIMED_EX:
{
    lbm_src_event_ume_ack_ex_info_t *ackinfo = (lbm_src_event_ume_ack_ex_info_t
        *)ed;
```

```
    if (opts->verbose) {
        printf("UME message reclaimed (ex) - sequence number %x (cd %p). Flags 0x%x
            ",
               ackinfo->sequence_number, (char*)(ackinfo->msg_clientd) - 1,
                   ackinfo->flags);
        if (ackinfo->flags & LBM_SRC_EVENT_UME_MESSAGE_RECLAIMED_EX_FLAG_FORCED) {
            printf("FORCED");
        }
        printf("\n");
    }
}
break;
```

Java API

```
case LBM.SRC_EVENT_UME_MESSAGE_RECLAIMED_EX:
    UMESourceEventAckInfo reclaiminfo = sourceEvent.ackInfo();
    if (_verbose > 0) {
        if (reclaiminfo.clientObject() != null) {
            System.out.print("UME message reclaimed (ex) - sequence number "
                             + Long.toHexString(reclaiminfo.sequenceNumber())
                             + " (cd "
                             +
                                 Long.toHexString(((Long)reclaiminfo.clientObject()).longValue()
                             + "). Flags 0x"
                             + reclaiminfo.flags());
        } else {
            System.out.print("UME message reclaimed (ex) - sequence number "
                             + Long.toHexString(reclaiminfo.sequenceNumber())
                             + " Flags 0x"
                             + reclaiminfo.flags());
        }
        if ((reclaiminfo.flags() &
            LBM.SRC_EVENT_UME_MESSAGE_RECLAIMED_EX_FLAG_FORCED) != 0) {
            System.out.print(" FORCED");
        }
        System.out.println();
    }
    break;
```

.NET API

```
case LBM.SRC_EVENT_UME_MESSAGE_RECLAIMED_EX:
    UMESourceEventAckInfo reclaiminfo = sourceEvent.ackInfo();
    if (_verbose > 0) {
        System.Console.Out.Write("UME message reclaimed (ex) - sequence number "
                                 + reclaiminfo.sequenceNumber()
                                 + " (cd "
                                 +
                                     ((uint)reclaiminfo.clientObject()).ToString("x")
                                 + "). Flags "
                                 + reclaiminfo.flags());
        if ((reclaiminfo.flags() &
            LBM.SRC_EVENT_UME_MESSAGE_RECLAIMED_EX_FLAG_FORCED) != 0) {
            System.Console.Out.Write(" FORCED");
        }
        System.Console.Out.WriteLine();
    }
    break;
```

8.2.5 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 (source)**
- **ume_retention_unique_confirmations (source)**
- **retransmit_retention_size_threshold (source)**
- **retransmit_retention_size_limit (source)**

8.2.6 Confirmed Delivery

The configuration option **ume_retention_unique_confirmations (source)** 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. This provides a form of receiver-pacing; the source will not be allowed to exceed Persistence Flight Size beyond receiving applications.

For example, a topic might have 2 receivers which are considered essential to keep up, and which should therefore contribute to flight size calculation. There might be any number of less-essential receivers which can be allowed to lag behind. In this case, **ume_retention_unique_confirmations (source)** would be set to 2, and the non-essential receivers would set **ume_allow_confirmed_delivery (receiver)** to 0.

Note

Smart Sources do not support delivery confirmation.

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);
}
```

8.2.7 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 Persistence 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 =
            (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;
}
```

8.2.8 Source Event Handler - Stability, Confirmation and Release

As shown in Source Event Handler 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 Persistence 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 reg 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 reg 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;
```

8.2.9 Mapping Your Message Numbers to Sequence Numbers

The C API function **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;
}
```

8.2.10 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 (context)**
- **ume_receiver_liveness_interval (context)**
- **ume_confirmed_delivery_notification (source)**
- **ume_user_receiver_registration_id (context)**
- **ume_session_id (context)**, **ume_session_id (source)**, **ume_session_id (receiver)**

Note

Smart Sources do not support liveness detection.

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 Router.
- 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.

8.3 Designing Persistent Receivers

Receivers are predominantly interested in RegID management and recovery management.

8.3.1 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 UM 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 UM. You can also use Session IDs to enable UM 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.

Receiver Message and Event Handler

The Receiver Message and Event Handler is an application callback, defined at receiver initialization, to deliver received messages to your application. 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 Persistence 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 =
                (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;
}
```

8.3.2 Recovery Management

Recovery management for failed and restarted receivers is fairly simple. UM requests any missed messages from the store(s) and delivers them to the restarted receiver. However, your application can override that default behavior either by configuring a **retransmit_request_maximum (receiver)** value, or by configuring a **ume_recovery_↩ sequence_number_info_function (receiver)** application callback, or both.

For example, let's say a source sends 7 messages with sequence numbers 0-6 which are stabilized at the store. A C-based receiver, configured with **retransmit_request_maximum (receiver)** set to 2, and an application callback **ume_recovery_sequence_number_info_function (receiver)**, consumes (and acknowledges) message 0, goes down, then restarts right after message 6.

During receiver registration, the **lbm_ume_rcv_recovery_info_ex_func_t** application callback is called with the following values in the passed-in structure **lbm_ume_rcv_recovery_info_ex_func_info_t** ∗info:

```
info->high_sequence_number == 6
info->low_rxreq_max_sequence_number == 4
info->low_sequence_number == 1
```

Where:

- **lbm_ume_rcv_recovery_info_ex_func_info_t::high_sequence_number** - the most recent message sent by the source,

- **lbm_ume_rcv_recovery_info_ex_func_info_t::low_rxreq_max_sequence_number** - high_sequence_↩ number (above) minus the number configured for **retransmit_request_maximum (receiver)** (2 in this example), and
- **lbm_ume_rcv_recovery_info_ex_func_info_t::low_sequence_number** - the first sequence number missed by the receiver after it went down.

Normally, UM would start delivering messages at 1, but **retransmit_request_maximum (receiver)** is set to 2, which overrides UM's normal behavior. So in this example, the first message delivered will be number 4.

Finally, the application can, at run-time, further override the starting sequence number. The callback function can modify the contents of the passed-in structure **lbm_ume_rcv_recovery_info_ex_func_info_t** ∗info; specifically it can update the **lbm_ume_rcv_recovery_info_ex_func_info_t::low_sequence_number** field. When the callback returns, UM examines that field to see if it was modified by the callback. If so, UM overrides the effect of **retransmit**↩ **_request_maximum (receiver)** and starts at the requested sequence number.

Notice that this design does not allow the callback to override the effect of **retransmit_request_maximum (receiver)** by setting the **lbm_ume_rcv_recovery_info_ex_func_info_t::low_sequence_number** field to its original value, 1 in this example. Upon return, UM will see the value unchanged, and will allow **retransmit_request_**↩ **maximum (receiver)** to override the starting sequence number. This is only an issue if *both* **retransmit_request**↩ **_maximum (receiver)** and **ume_recovery_sequence_number_info_function (receiver)** are used. If the application wants to use the sequence number remembered by the store, it should not configure **retransmit_request_**↩ **maximum (receiver)**.

8.3.3 Duplicate Message Delivery

In a distributed system, it is not possible to *guarantee* "once-and-only-once" delivery of messages in the face of unpredictable system or component failure. Regardless of the algorithms and handshaking, there is always the possibility of messages sent that are never received, as well as messages received and then received again if the receiving application fails and restarts.

UM's persistence design is based on the principle of being close to once-and-only-once, but when that is not possible, UM prefers to fail on the side of duplicate message delivery. Due to other design goals (low latency and high throughput), the possibility of receiving duplicate messages is significant after an application failure and restart.

It is therefore important for persistent applications to be designed to tolerate duplicate message reception, either by making message processing idempotent, or by including logic in the receiving application to detect duplicates and only process the messages which have not been previously processed.

To assist the application in implementing "de-duplication", 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 "live" 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.

Informatica recommends that you always check the data or other message properties of messages with the retransmit flag set to be sure the message has not been already processed. Relying on UM sequence numbers is not a 100% reliable method for detecting duplicate messages.

8.3.4 Setting Callback Function to Set Recovery Sequence Number

Whereas the UM persistence design attempts to choose the correct starting sequence number for a recovering receiver, there are cases where the application wishes to override UM's choice.

The sample code below demonstrates how to use the recovery sequence number info function to determine the stored message with which to restart a receiver. This example 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 **lbm_ume_rcv_recovery_info_ex_func_info_t::low_sequence_number** to the **lbm_ume_rcv_recovery_info_ex_func_info_t::high_sequence_number** plus one.

C API

```
lbm_ume_rcv_recovery_info_ex_func_t cb;

cb.func = ume_rcv_seqnum_ex; / * declared below */
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;
        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;
        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;
    }
}
```

8.3.5 Persistence Message Consumption

Receivers use message consumption, defined as message deletion, to indicate that UM 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, and optionally to the source if you configure the source for delivery confirmation.

In many applications, the message receiver application callback will fully process the received message. When the application callback returns, the message should be deleted and acknowledged.

However, there are other application designs where a received message cannot be fully processed inside the receiver application callback. For example, the message might need to be passed to a worker thread for longer-term processing. Or the acknowledgement must be delayed until some other event happens, like a handshake with another application. In these cases, the message deletion and/or message acknowledgement must not be done when the receiver callback returns.

Finally, for high-throughput applications, an application can completely suppress the acknowledgement of each individual message in favor of acknowledgement batching (acknowledging multiple messages in one operation). This is done to reduce the per-message overhead. Note that acknowledgement batching increases the chances that a restarted application will receive duplicate messages (messages that had been previously process but not yet acknowledged). See Duplicate Message Delivery for more information.

8.3.6 Immediate Message Consumption

In many applications, the message receiver application callback will fully process the received message. When the receive callback returns, the message should be deleted and acknowledged. This is handled differently between the C API vs. the Java and .NET APIs.

C API

The default behavior for a C receiver application callback is for the message to be deleted and acknowledged when the receiver callback returns. No special coding is needed for this use case.

Java and .NET

With Java and .NET, the UM library is not able to differentiate between a message that is passed to a different part of the application vs. a message which is simply dereferenced for eventual garbage collection. So the default behavior of the UM library is different – it is assumed that the message should *not* be deleted and acknowledged when the receiver application callback returns. Instead, the application is expected to explicitly dispose of received messages when processing is complete.

In the case where message processing is completed in the receiver callback, the application must call the "dispose()" method of the message object before returning. This triggers acknowledgement as well as cleanup of the message's resources.

8.3.7 Delayed Message Processing

There are application designs where a received message cannot be fully processed inside the receiver application callback. For example, the message might need to be passed to a worker thread for longer-term processing. Or the acknowledgement must be delayed until some other event happens, like a handshake with another application.

This is handled differently between the C API vs. the Java and .NET APIs.

C API

In the C API, the application's receiver callback function must call the **lbm_msg_retain()** function for the received message. This suppresses the automatic deletion of the received message when the receiver callback returns, and allows the message buffer to be handed to some other part of the application for processing and deletion at a later time.

When the application subsequently completes all processing of the message and is ready for the message to be deleted and acknowledged, it calls **lbm_msg_delete()**.

Java and .NET

With Java and .NET, the UM library assumes that the message should *not* be deleted and acknowledged when the receiver application callback returns. The callback can simply pass the message to some other part of the application for subsequent processing.

When the application has completed all processing on the message, the message's "dispose()" method should be called. This releases resources held by the object and also triggers the acknowledgement.

8.3.8 Batching Acknowledgments

For high-throughput applications, it is often desired to reduce the per-message overhead. Sending acknowledgements to the Store and optionally to the source normally involves multiple socket operations, which can limit the maximum sustainable throughput of a persistent receiver.

A significant reduction in per-message overhead can be achieved by batching acknowledgements. In this use case, the sending of acknowledgements is delayed until multiple messages have been received and processed. Then an

acknowledgement is sent which covers all messages processed so far.

Warning

> While ACK batching provides significant improvements in receiver throughput, it also increases the probability that a failed and restarted receiver will be sent duplicate messages (i.e. messages that the application has already received and processed).

ACK Batching can be done implicitly or explicitly. For implicit ACK batching, use the configuration options **ume_use_ack_batching (receiver)** and **ume_ack_batching_interval (context)**. Note that implicit ACK batching also supports out-of-order acknowledgements. See ACK Ordering.

Explicit ACK batching gives the application precise control over when acknowledgements are sent via API calls. This mode of operation is enabled with the **ume_explicit_ack_only (receiver)** configuration option. If enabled, acknowledgements are only sent as a result of the application explicitly calling an API. This allows the application to use application-level knowledge to optimize when to send acknowledgements, potentially minimizing the time that processed messages are left unacknowledged (and therefore minimizing the number of potential duplicate messages).

See **lbm_ume_ack_send_explicit_ack()** and **lbm_msg_ume_send_explicit_ack()** for the C API. See **com::latencybusters::lbm::LBMMessage::sendExplicitAck()** for Java and .NET. See Explicit Acknowledgments for details on explicit ACKs.

8.3.9 ACK Ordering

The Persistent Store does not support "out of order" acknowledgement of messages. If the Store receives an acknowledgement of sequence number N, that implicitly acknowledges all sequence numbers less than N. If a receiving application has the ability to complete processing of messages out of order, it must ensure that an acknowledgement is sent for a given message until all previously-received messages have been completely processed.

Normally, the only way that a receiving application *can* process messages out of order is to retain those messages and complete processing of them outside of the receiver application callback function. This normally requires "retaining" the messages so that they aren't deleted (and therefore acknowledged) automatically when the receiver callback returns. In this usage, when a message is completely processed, that message is deleted by the application, triggering the acknowledgement of that message. However, if the application design allows those messages to be processed out of order, then the risk exists that the acknowledgement of a given message will implicitly acknowledge previous message which have not been completely processed. This will prevent those incompletely processed messages from being recovered if the receiving application fails and restarts.

ACK Batching can provide a solution, implicitly or explicitly.

The implicit form of ACK batching provides, as a convenience, the ability to postpone the sending of a message ACK until all previous received messages have also been processed. When the UM context wakes up every **ume_ack_batching_interval (context)** milliseconds, it checks for unacknowledged messages that have been deleted, either implicitly from the receiver callback returning, or explicitly by API calls to retain and then delete the message. UM will only acknowledge up to the highest *continuous* sequence number.

For example, let's say the application deletes messages with sequence numbers 0, 1, 5, 2, 4. Messages 3 and 6 are still being processed. If the context wakes up at this point, it will send an acknowledgement for sequence 2. If the application fails at this point and restarts, the Store will re-send messages 3, 4, 5, and 6. The receiving application must handle the fact that 3 and 6 were incompletely processed, whereas 4 and 5 were completely processed (see Duplicate Message Delivery).

Instead of using implicit batching for this, the application can be coded to use Explicit Acknowledgments. However, in this case, the application has the responsibility to implement a similar algorithm as the implicit ACK batcher described above. I.e. even though the messages 4 and 5 were fully processed, the application would need to postpone sending an acknowledgement until message 3 is also completed, at which point a single acknowledgement for sequence 5 can be sent.

8.3.10 Explicit Acknowledgments

UM supports Explicit acknowledgement which suppresses UM's default acknowledgement behavior, allowing your application complete control of message consumption notification.

There are two common use cases for Explicit Acknowledgements:

- Deferred Acknowledgement.
- Application-level ACK batching.

Deferred Acknowledgement means that the receiving application is not able to fully process a message within the message receiver application callback. For example, the message may require processing in a separate thread. By default, UM will acknowledge a persisted message when the receiver callback returns.

Application-level ACK batching means that the application chooses not to acknowledge every received message. Instead, it implements its own logic to decide which messages to acknowledge. Note that acknowledging a given message implicitly acknowledges all earlier messages. For example, acknowledging messages 5, 10, and 15 tells the Store that *all* messages 0-15 are acknowledged.

Also note that this imposes the restriction that messages be acknowledged in ascending order. See ACK Ordering for more information.

Explicit acknowledgement is enabled using the configuration option **ume_explicit_ack_only (receiver)**.

8.3.11 Object-free Explicit Acknowledgments

When using explicit ACKs, you can extract ACK information from messages. This allows the received message buffer to be deleted when the receiver callback is done, while still allowing the application to save the ACK structure for persistent acknowledgement to the Store at a future time. This can improve receiver performance when used with the **Receive Buffer Recycling** feature to reduce the per-message use of dynamic memory (malloc/free) with a persistent receiver. Extracting ACKs can also additionally improve performance of Java and .NET applications by allowing the use of **Zero Object Delivery**.

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);
  defer_ack(ack); / * Pass the "ack" to another thread or work queue. */
...
  return 0;
}

int worker()
{
  lbm_ume_rcv_ack_t *ack = NULL;
...
  ack = get_deferred_ack(); / * Get "ack" that was saved above. *//

  * Some applications improve throughput by not ACKing every message. */
```

```
  if (ack_this_message) {
    lbm_ume_ack_send_explicit_ack(ack, msg->sequence_number);
  }

  lbm_ume_ack_delete(ack); / * Extracted ack *must* be deleted. */
...

}
```

JAVA API or .NET API

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

  ack = msg.extractUMEAck();
  defer_ack(ack); / * Pass the "ack" to another thread or work queue. */
...
  return 0;
}

int worker()
{
  UMEMessageAck ack;

  ack = get_deferred_ack(); / * Get "ack" that was saved above. *//

  * Some applications improve throughput by not ACKing every message. */
  if (ack_this_message) {
    ack.sendExplicitAck(msg.sequenceNumber());
  }
  ack.dispose(); / * Extracted ack *must* be deleted. */
}
```

8.4 Designing Persistent Stores

As mentioned in Persistent Store Concept, the 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 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.

A source does not send data to the store and then have the store forward it to the receivers. In UM, the source sends to receivers and the stores in parallel. See Persistence Normal Operation. Thus, UM 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) in a Quorum/Consensus arrangement.

Receivers, on the other hand, do not need to be configured with store information a priori. The source provides store information to receivers via a Source Registration Information (SRI) message after the source registers with a store. Thus the receivers learn about stores from the source, without needing to be configured themselves. Because receivers learn about the store or stores with which they must register via a SRI record, the source must be available to receivers. However, the source does not have to be actively sending data to do this.

8.4.1 Store Log File

The store daemon generates log messages that are used to monitor its health and operation. You can configure these to be directed to "console" (standard output) or a specified log "file", via the <log> configuration element. Normally "console" is only used during testing, as a persistent log file is preferred for production use. The store does not over-write log files on startup, but instead appends them.

8.4.2 Store Rolling Logs

To prevent unbounded disk file growth, the store supports rolling log files. When the log file rolls, the file is renamed according to the model:
CONFIGUREDNAME_PID.*DATE*.*SEQNUM*
where:

- *CONFIGUREDNAME* - Root name of log file, as configured by user.
- *PID* - Process ID of the store daemon process.
- *DATE* - Date that the log file was rolled, in YYYY-MM-DD format.
- *SEQNUM* - Sequence number, starting at 1 when the process starts, and incrementing each time the log file rolls.

For example: `umestorelog_9867.2017-08-20.2`

The user can configure when the log file is eligible to roll over by either or both of two criteria: size and frequency. The size criterion is in millions of bytes. The frequency criterion can be daily or hourly. Once one or both criteria are met, the next message written to the log will trigger a roll operation. These criteria are supplied as attributes to the <log> configuration element.

If both criteria are supplied, then the first one to be reached will trigger a roll. For example, consider the setting:

```
<log type="file" size="23" frequency="daily">store.log</log>
```

Let say that the log file grows at 1 million bytes per hour. At 11:00 pm, the log file will reach 23 million bytes, and will roll. Then, at 12:00 midnight, the log file will roll again, even though it is only 1 million bytes in size.

Note

> The rolling logs cannot be configured to automatically overwrite old logs. Thus, the amount of disk space consumed by log files will grow without bound. The user must implement a desired process of archiving or deleting older log files according to the user's preference.

8.4.3 Quorum/Consensus Store Usage

To provide the highest degree of resiliency in the face of failures, UM provides the Quorum/Consensus failover strategy which allows a source to provide UM with a number of stores to be used at the same time. Multiple stores can fail and messaging can continue operation unhindered as long as a majority of configured stores are operational.

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 UM, 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. UM 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 UM 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.

8.4.4 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.

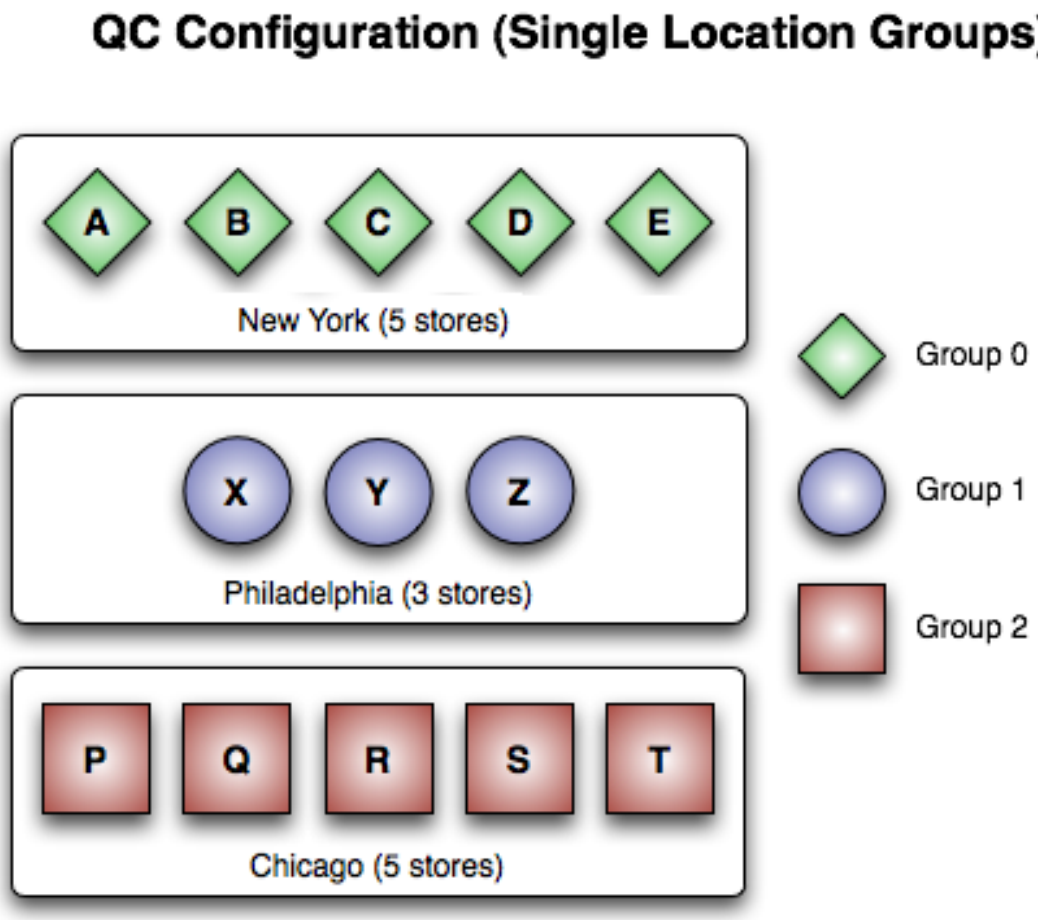
- The intragroup behavior specifies the requirements needed to stabilize a message among the stores within a group. A message is stable for the group once it is successfully stored at a quorum (majority) of the group's stores or successfully stored in all the stores in the group.

- The intergroup behavior 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, all groups, or all active 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 figure displays a 3-group Quorum/Consensus configuration with each group in a different location. A message is considered stable when it has been successfully stored at a quorum of stores in all the active groups.

Quorum/Consensus - Single Location Groups



The source application's UM configuration file appears below.

```
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 all-active
```

8.5 Persistent Fault Recovery

Recovery from source and receiver failure is the real heart of persistent 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 UM to make their applications recover from failure in graceful ways.

Late Join is the mechanism of persistent recovery as well as an UM streaming feature. If Late Join is turned off on a source (**late_join (source)**) or receiver (**use_late_join (receiver)**), it also turns off persistent recovery. In order to control Late Join behavior, UM 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.

8.5.1 Persistent 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 UM 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, UM 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 (source)** 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 Persistence Events delivered to the application to mimic the behavior of restricting a source to having only one unstable message at a time.

8.5.2 Persistent 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.

UM 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 UM persists receiver state at the store, receivers request this state from the store as part of re-registration and recovery. Receiving applications experiencing unrecoverable loss can potentially retrieve missed messages from the stores by deleting and recreating the receiver object.

The actual sequence number that a receiver uses as the first topic level message to resume reception with is called the "low sequence number". UM 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, UM 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 (receiver)** 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, UM 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.

8.6 Callable Store

It is possible for an application to start an instance of the store to run as an independent set of threads within the application process. However, there are several restrictions:

1. The application may not make use of messaging. I.e. an application which intends to start a store instance must not create contexts, sources, or receivers, or make any use of UM except starting (and optionally stopping) the store. For applications that need to use messaging, it is suggested that the application create a child process from which to invoke the store. The parent process can then use messaging freely. See the example program `umestored_example.c` for an example of how this can be done.
2. Only a C API is provided at this time. Two API functions are available: **umestored_main()** to start the store threads running, and **umestored_main_shutdown()** to request the store threads to stop gracefully.
3. The **umestored_main()** API will not return until the store exits, either by processing a signal, or by the application calling **umestored_main_shutdown()**. When **umestored_main()** does return, the store is in a safe state for the application to exit.
4. Only a single instance of the store may be started. This means that an application may not have two stores running concurrently, and it also means that an application may not start a store, shut it down, and then start it again. The store API is "single use".
5. The application may not set signal handlers for SIGPIPE, SIGUSR1, SIGINT, or SIGTERM. The store uses those signals. For applications that need to handle those signals, it is suggested that the application create a child process, as mentioned above (#1).

For an example of how to use the **umestored_main()** API, see the example program `umestored_example.c`. Note that while the callable store APIs are usable on all supported platforms, this example program is restricted to Linux due to its use of `prctl()`, a Linux-only function.

Chapter 9

Persistence Fault Tolerance

9.1 Message Loss Recovery

Persistence offers the following message recovery mechanisms:

Method	Product	Transports	Description
Negative Acknowledgments (N↩ AKs)	UMS, UMP, UMQ	LBT-RM, LBT-RU	Recovers lost transport datagrams from the source which may contain many small topic messages or fragments of a large message. Receivers send unicast NAKs to the source for missed transport datagrams. Source retransmits datagrams over the configured UM transport.
Late Join	UMS, UMP, UMQ	All	Retransmits messages via unicast to receivers joining the stream after the messages were originally sent. See **Using Late Join**.
Durable Receiver Recovery	UMP, UMQ	All	Recovers messages persisted while a durable receiver was off line. UM initiates recovery when a durable receiver joins a persistent stream. The receiver then requests retransmission from the store starting with the low sequence number, defined as the last message it acknowledged to the store plus one. The store unicasts retransmissions. See Persistent Receiver Recovery.

Method	Product	Transports	Description
Off Transport Recovery	UMS, UMP, UMQ	All	Recovers lost topic messages. Receiver detects lost sequence number and requests retransmission from the source or persistent stores (if applicable). UM unicasts retransmissions. See **Off-Transport Recovery (OTR)**.
Proactive Retransmissions	UMP, UMQ	All	Recovers lost messages never received by the store or never acknowledged by the store. Operates independently of any receivers. Source unicasts retransmissions. See Proactive Retransmissions.

9.2 Configuring for Persistence and Recovery

Deployment decisions play a huge role in the success of any persistent system. Configuration in UM 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 for the following:

- Source Considerations
- Receiver Considerations
- Store Configuration Considerations

9.2.1 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. UM provides a hard limit on the memory usage. When exceeded, UM delivers a Forced Reclamation event. 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 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.

9.2.2 Receiver Considerations

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

Acknowledgement Generation

Persistent receivers send 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 (receiver)** can be used.

Controlling Retransmission

Persistent receivers 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 (receiver)**, request everything at once. The value of the low sequence number (Persistent 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 (receiver)**. The timeout used between requests, if the retransmission does not arrive, is totally controllable with **retransmit_request_interval (receiver)**. And the total time given to recover all messages is also controllable.

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 UM 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 UM.

9.2.3 Store Configuration Considerations

UM stores have numerous configuration options. See Configuration Reference for Umestored for details.

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.

Disk vs. Memory

As mentioned previously in Persistent Store Concept, 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.

UM stores provide settings for controlling memory usage and for caching messages for retransmission in memory as well as on disk. 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.

Activity Timeouts

UM 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.

Recommendations for Store Configuration

The following conditions allow sources to continue to send messages:

- Quorum - Completed registration of a quorum of stores within at least one group. This is affected by group definitions, plus intragroup and intergroup stability settings. See also Persistent Store Concept.
- Flight Size - Maximum number of messages sent but not stable which is determined by store group definitions, intragroup and intergroup stability settings and delivery confirmation setting. See also Persistence Flight Size.

Configure your stores to address the failure cases you believe are more probable and from which you want to recover. For example, if a particular store group persists topics of higher importance, you may want to increase the number of stores in that group to maintain quorum in the face of a store failure. Or if a particular location has a higher incidence of failures than other locations, you may want to add additional stores in other locations.

Although many different conditions and requirements can apply to the configuration of persistent stores, Informatica recommends the following best practices:

1. Minimum of 3 stores - Requiring a minimum of 3 stores needed for quorum in a single store group is optimal. Using 5 stores, for example, in a group allows sources to keep sending in the face of the loss of up to 2 stores.
2. Multiple store groups - When using multiple store groups, Informatica recommends using at least 3 stores in each group.
3. Set Intergroup Stability to all-active. This setting for **ume_retention_intergroup_stability_behavior (source)** provides a more immediate evaluation of your store configuration. Active groups must have at least a quorum of active stores, registered with the source and sending stability acknowledgements for persisted messages. By default, if a store becomes unresponsive, a store group could lose quorum and therefore messages in-flight cannot be stabilized by the unresponsive store's group until the store's **ume_store_activity_timeout (source)** expires and the store restarts. However, with all-active, the source does not wait for the unresponsive store's **ume_store_activity_timeout (source)** to expire. The source removes the unresponsive store's group from the list of stores from which the source uses to determine that messages in-flight are stable. An inactive store with a running activity timeout does not impede message stabilization.

Store Configuration Practices to Avoid

Informatica does not support the following store configuration practices:

- Do not use multiple store groups of one store each. Recovery does not work well in this configuration because it allows sources to resume sending as soon as it has registered with a single store, and if that store is not fully up-to-date, this can lead to message loss for receivers.
- Do not use backup stores. The configuration option, **ume_store_group (source)** allows you to identify a store group and its size in number of stores. Setting the group size in this option to a number of stores less than the number of stores configured with the **ume_store (source)** option can lead to messages that were reported stable to the source being unavailable for receivers to recover, in the event that multiple stores became unresponsive and were replaced by backup stores. For example, setting **ume_store_group (source)** with a size of 3 stores, but configuring ume_store with 5 stores is not supported.

9.3 Persistence Proxy Sources

By default, UM expects persistent sources to be running concurrently with persistent receivers. If a source exits, any persistent receivers will disconnect from that source's transport and will wait for the source to come back. More significantly, if a new receiver starts while the source is absent, the receiver will be unable to discover the stores where the old source's previous messages are stored. So that late-joining receiver will not recover messages until the source finally restarts.

The Proxy Source feature allows you to configure stores to automatically continue sending the source's topic advertisements which allow new receivers to join the source's transport session and request Source Registration Information (SRI) to register with the store and request retransmissions. After the source returns, the store automatically stops acting as a proxy source. Stores can be located across a UM Router or within the same LAN as the failed 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, each one corresponding to a previous instance of a real source.

Note that proxy sources do introduce extra network and CPU loading, so proxy sources should only be enabled if their functionality is needed.

9.3.1 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 (source)** 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.

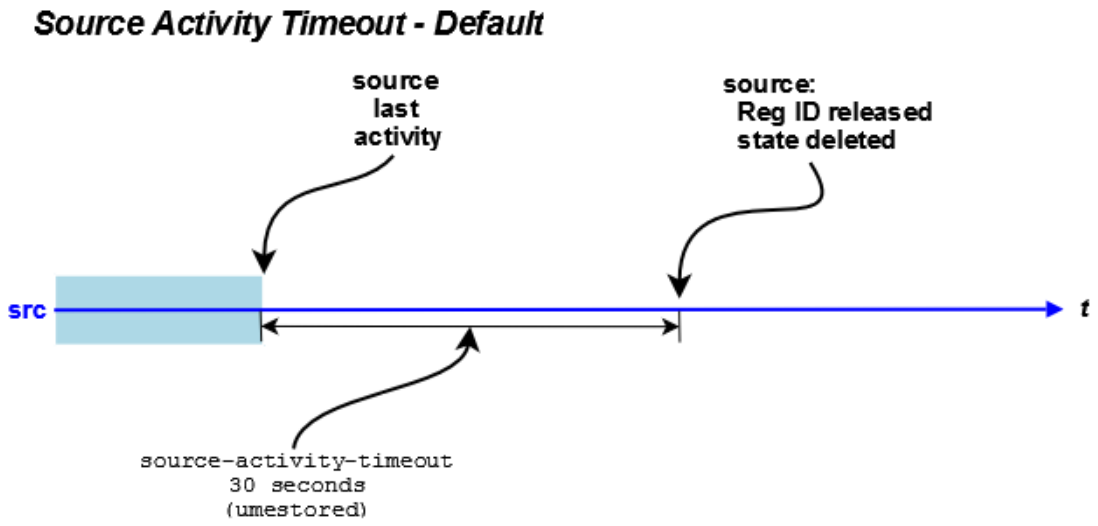
If the store running the proxy source fails, the other stores in the Quorum/Consensus group detect a source failure again and elect a new store to initiate a proxy source.

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

9.3.2 Activity Timeout and State Lifetimes

UM 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.

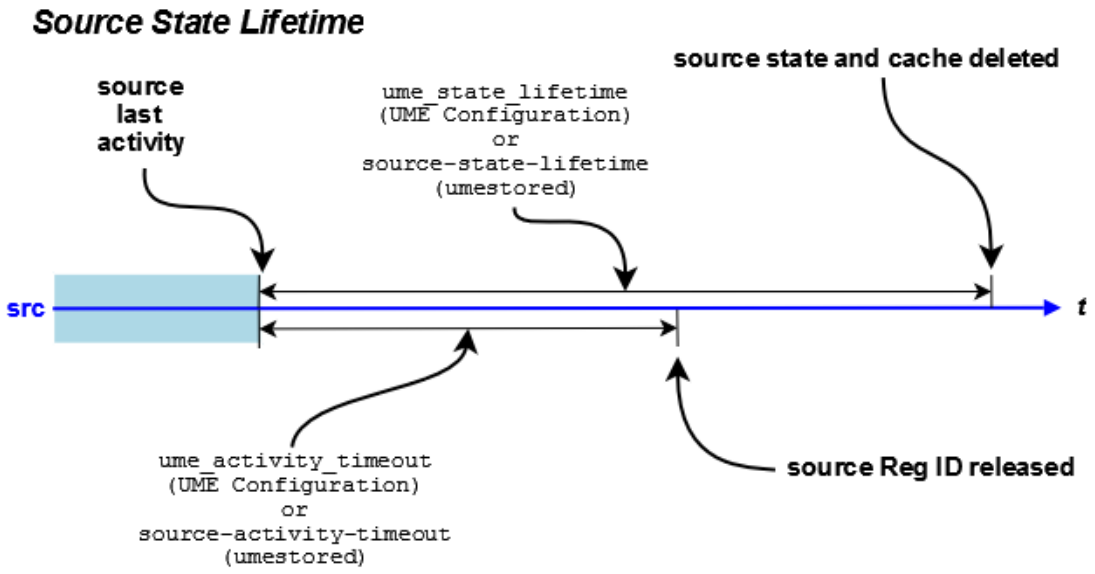
The **ume_activity_timeout (source)** and **ume_activity_timeout (receiver)** options determine how long a source or receiver must be inactive before a store allows another source or receiver to register using that RegID. This prevents a second source or receiver from stealing a RegID from an existing 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.



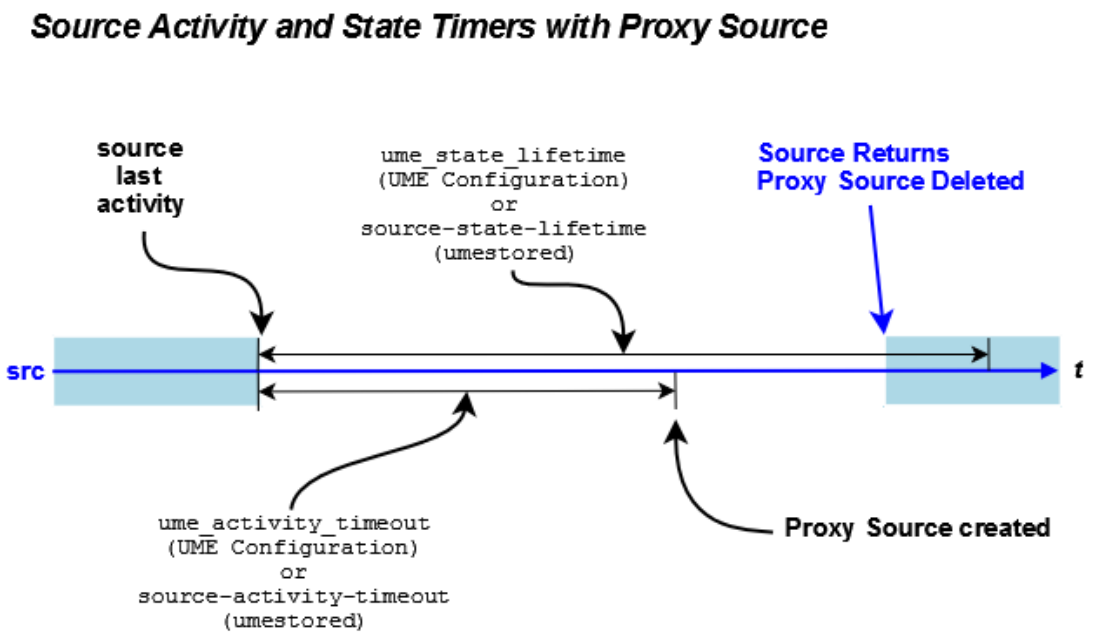
In addition to the activity timeout, you can also configure sources and receivers with a state lifetime timer using the following options.

- **ume_state_lifetime (source)**
- **ume_state_lifetime (receiver)**
- The topic's ume-attributes options, source-state-lifetime and receiver-state-lifetime.

The **ume_state_lifetime (source)** and **ume_state_lifetime (receiver)** options, when used in conjunction with the **ume_activity_timeout (source)** and **ume_activity_timeout (receiver)** options, determines at what point UM removes the source or receiver state files. UM does not check the state lifetime until the activity timeout expires. The following diagram illustrates this behavior:



If you have enabled the Proxy Source option, the **ume_activity_timeout (source)** triggers the creation of the proxy source. The following diagram illustrates this behavior:



9.3.3 Enabling the Proxy Sources

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 (source)**.
- Configure the stores in the umestored XML configuration file with the Store Element Option, allow-proxy-source.

9.3.4 Proxy Source Elections

When multiple stores in a Quorum/Consensus configuration notice the loss of a registered source (expiration of the source's **ume_activity_timeout (source)**) configured for proxy sources, only one of the stores needs to create a proxy source to continue sending topic advertisements.

The proxy source election process determines which store creates the proxy source. Each store starts by waiting a randomized amount of time based on its proxy-election-interval option setting. The store creates a proxy source if it has not received a persistent registration request (PREG) from a proxy on a different store. The proxy source then sends a PREG containing a unique random value to the other stores. This value determines which store deletes it's proxy source in the case that any two stores independently determine they should create a proxy source. The nature of the random values ensures that only one store within the Q/C group or configuration of groups keeps its proxy source.

9.3.5 Proactive Retransmissions

Proactive Retransmissions, which is enabled by default, address two types of loss:

- loss of message data between the source and a store
- loss of stability acknowledgments (ACK) between the store and the source

The store sends message stability acknowledgments to the source after the store persists the message data.

With Proactive Retransmissions, the source maintains an unstable message queue for those messages sent but not acknowledged by the store. The source checks this queue at the **ume_message_stability_timeout (source)**. If a message in this queue exceeds its **ume_message_stability_timeout (source)**, the source retransmits the message and puts it back on the unstabilized message queue, restarting the message's **ume_message_stability_timeout (source)**.

The source continues to retransmit and check the message's stability timeout until the **ume_message_stability_lifetime (source)** expires or it receives a stability acknowledgment from the store. If the source has not received a stability acknowledgment when the **ume_message_stability_lifetime (source)** expires, the source sends a Store Message Not Stable source event notification to the application. When the store discards the message because it has not met stability requirements, the store sends a Store Forced Reclaim source event notification to the application.

To disable Proactive Retransmissions, set **ume_message_stability_timeout (source)** to 0 (zero). As a result, sources do not create an unstable message queue.

The following applies whether you enable or disable Proactive Retransmissions.

- The store does not discard duplicate messages, but rather always responds to duplicate, retransmitted messages by sending stability acknowledgments even if the message is already stable.

- If the store has marked the message unrecoverably lost and receives a duplicate message from the source, the store sends the source a negative stability acknowledgment (NAK), which induces the source to remove the message from its unstabilized message queue. A stability NAK is identical to a stability ACKs except that it has a NAK flag set.

Chapter 10

Persistence Man Pages

10.1 umestored Man Page

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

Description

Persistent Store 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 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 (not recommended unless absolutely necessary).

Exit Status

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

10.2 umestoreds Man Page

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

Description

Persistent Store 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 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 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.

Chapter 11

Configuration Reference for Umestored

The operating parameters for umestored come from an XML configuration file that must be supplied on the Man Pages. 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 with the umestored XML configuration file, which Ultra Messaging reads at start up.

Note: umestored implements only persistent streaming functionality. See the Guide to Queuing for information about configuring queues.

The umestored XML configuration file for persistence 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.

High Level Store Configuration File:

```
<ume-store version="1.3">
  <daemon>
    Daemon configuration options
  </daemon>
  <stores>
    <store attributes>
      <topics>
        <topic attributes>
          <ume-attributes>
            <option attributes/>
          </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
</ume-store>
```

11.1 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.
uid	User ID (uid) for daemon process (if started as root)	Daemon retains starting uid
pidfile	Pathname for daemon process ID (pid) file	No pidfile
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
xml-config	Pathname for UM XML 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: ∗:15304	No web monitor
daemon-monitor	configuration for **Daemon Statistics**.	

11.1.1 Log Element

The Log Element defines how the store daemon writes log messages during operation.

The following table gives attributes for the <log> element:

Attribute	Description	Default Value
type	One of "file", "console".	"console"
size	Number of millions of bytes of file size to roll log file. E.g. a value of 1 rolls after 1000000 bytes. Maximum value is 4000. Value of 0 disables rolling by file size. Only applicable for type="file".	0
frequency	Frequency by which to roll log file. Choices are daily - Roll log file at midnight. hourly - Roll log file after approximately an hour, but is not exact and can drift significantly over a period of time. test - For internal Informatica testing and should not be used. disable - Do not roll log file by frequency. Only applicable for type="file"	disable

11.1.2 Daemon-monitor Element

The daemon-monitor element configures the store daemon for **Daemon Statistics**.

See Store Daemon Statistics Configuration for an example of configuring.

The following table gives attributes for the <daemon-monitor> element:

Attribute	Description	Default Value
topic	topic name for used to publish daemon statistics.	umestore.monitor

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

Tag	Description	Default Value
lbm-config	Optional LBM configuration file for the monitoring context.	Defaults to LBM settings set by the daemon element's lbm-config element.
publishing-interval	optional element which sets the interval for message publication.	Defaults to not publishing.
remote-snapshot-request	Controls if the daemon will respond to requests from monitoring applications. See **Daemon Statistics Requests**.	Defaults to disabled.
remote-config-changes-request	Controls if the daemon will respond to change requests from monitoring applications. See **Daemon Statistics Requests**.	Defaults to disabled.

11.1.3 Publishing-interval Element

Element which sets the publishing intervals for desired message types. The desired message types must be listed as <group> sub-elements of the <publishing-interval> element.

See Store Daemon Statistics Configuration for an example of configuring.

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

Tag	Description	Default Value
group	Identifies the grouping of message type(s) and their publishing intervals.	Any grouping of message type(s) not listed default to not publishing.

11.1.4 Group Element

The group element identifies the grouping of message type(s) and their publishing interval.

The following table gives attributes for the <group> element:

Attribute	Description	Default Value
name	one of the following: • "default" - sets a default interval for all message types. • "store" - sets the interval for messages of type **umestore_store_dmon_stat_msg_t**. • "source" - sets the interval for messages of type **umestore_repo_dmon_stat_msg_t**. • "receiver" - sets the interval for messages of type **umestore_rcv_dmon_stat_msg_t**. • "disk" - sets the interval for messages of type **umestore_disk_dmon_stat_msg_t**. • "config" - sets the interval for messages of types umestore_*_dmon_config_msg_t. • "memory" - sets the interval for messages of type **umestore_smart_heap_dmon_stat_msg_t**.	Required attribute (no default).
ivl	Time interval in seconds.	Required attribute (no default)

11.2 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>
```

11.2.1 Store Element

The Store Element contains information about an individual 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 the <store> element:

Attribute	Description	Default Value
name	Name that identifies log messages for this store in the umestored log file.	None—this is a required attribute
port	TCP port where umestored should listen for connection requests to this store.	None—this is a required attribute
interface	Specifies the IP address over which umestored accepts connection requests for this store. You can specify a single IP address, such as 10.29.3.16, or a range of addresses, 10.↩ 29.3.16/25. See also Identifying Persistent Stores	0.0.0.0 (INADDR_ANY)

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
ume-attributes	A container for option elements. See Options for a Store's ume-attributes Element for more information.	None
publishing-interval	optional element which sets the interval for message publication.	Defaults to not publishing.

Options for a Store's ume-attributes Element

Inside a '<ume-attributes>' section, options are set with one or more '<option ...>' elements. Each '<option ...>' element contains a 'type' attribute, a 'name' attribute, and a 'value' attribute. The 'type' attribute identifies the scope of the option (context, receiver, source, or store), the 'name' attribute identifies the individual option being set, and the 'value' attribute supplies the value.

Options for UM

Options with a type of "lbm-source", "lbm-receiver", or "lbm-context" are UM configuration options, which the store passes to the UM library.

For example, the option element:
<option type="lbm-context" name="transport_lbtrm_receiver_socket_buffer" value="1048576"/>
sets the UM configuration option **transport_lbtrm_receiver_socket_buffer (context)** (receiver socket buffer size for LBT-RM) to 1 megabyte.

See the UM Configuration Guide for the full list of UM configuration options.

Note

Some UM options specify interfaces, which can be done by supplying the device name of the interface. Special care must be taken when supplying device names. See **Interface Device Names and XML** for details.

Store Options

Store options without a type attribute or those explicitly given a type attribute of 'store' simply configure the store itself.

For example, the option element:
`<option type="store" name="disk-cache-directory" value="cache"/>`
sets the store's disk cache directory.

The following table gives options allowed for a store element. Use the 'store' option type for these options.

Option	Description	Default Value
disk-cache-directory	Pathname for disk store message cache directory. Must be between 1 and 230 characters long.	umestored-cache
disk-state-directory	Pathname for disk store state directory. Must be between 1 and 230 characters long.	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. Restricted to 128 characters in length, and may contain only alphanumeric characters, hyphens, and underscores. A store runs in its own context, so the store's context name can be used to identify the store. UM automatically resolves store names, which can facilitate persistent operation across the UM Router. A context name must be unique across the entire network and not be the same as any context names used in a UMM XML configuration. See also Identifying Persistent Stores	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

11.2.2 Topics Element

The Topics element is a container element for all the topics persisted by the UM 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>
```

11.2.3 Topic Element

The Topic Element defines an individual topic persisted on the 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. **The regexp selection is deprecated.**	direct

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

Options for a Topic's ume-attributes Element

As with Options for a Store's ume-attributes Element, options for a topic's ume-attributes element can set both UM configuration options as well as store configuration.

The following table gives options allowed for a topic element. Use the store Option Type for these options.

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

Option	Description	Default Value
repository-type	Specifies how messages should be retained by the store. Possible values: • "no-cache" does not retain messages, only state information. • "memory" retains messages only in the (presumably volatile) main memory of the store. • "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. • "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 before the repository starts to delete old messages. Pertains to a memory store or the memory cache of a disk or reduced-fd repository. For RPP repositories, this value only includes message payload. Also for RPP, the source may optionally override this value with a value less than or equal with the source configuration option **ume_repository_↩ size_threshold (source)**. (units↩ : bytes)	25165824 (24 MB)

Option	Description	Default Value
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. Pertains to a memory store or the memory cache of a disk or reduced-fd repository. For RPP repositories, this value only includes message payload. Also for RPP, the source may optionally override this value with a value less than or equal with the source configuration option **ume_repository_size_limit (source)** (units: bytes)	50331648 (48 MB)
repository-age-threshold	For topics with a repository-type of memory, disk or reduced-fd, specifies how long the repository keeps a message available. Pertains to a memory store or the memory cache of a disk or reduced-fd repository. The repository reclaims space used to store messages that exceed this threshold. 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) This option is deprecated.	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 recommends a value of 200 times the number of expected receivers per topic. (units: async callbacks)	16 callbacks

Option	Description	Default Value
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. For RPP, the source may optionally override this value with a value less than or equal with the source configuration option **ume_repository_disk_file_size_limit (source)**. (units: bytes)	104857600 (100 MB)
repository-disk-file-preallocate	For topics with a repository-type of disk, if set to 1, UM 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)	1024000
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 (source)**) the store does not release the source's RegID and UM elects a proxy source. If neither proxy sources nor **ume_state_lifetime (source)** are configured, the store also deletes the source's state and cache. Can be overridden by **ume_activity_timeout (source)**. See also Persistence 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 (source)** for the source. The store uses whichever is shorter. See also Persistence 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 (source)**. See also Persistence Proxy Sources. (units: milliseconds)	30000 (30 seconds)
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 (receiver)** for the receiver. The store uses whichever is shorter. See also Persistence Proxy Sources. (units: milliseconds)	0 (zero)
source-check-interval	Specifies how often a store will check for activity of sources and receivers. (units: milliseconds)	750 (750 milliseconds)

Option	Description	Default Value
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)	3000 (3 seconds)
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)	2147483647 (rollback starts at beginning of stream)
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)	60,000 (60 seconds)
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 stores (especially memory stores) significantly, but introduces a delay between when a message is actually stable in the store and when the source is notified of message stability. (units: milliseconds)	200 (200 milliseconds)

Option	Description	Default Value
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 stores (especially memory stores) significantly, but introduces a delay between when a message is actually stable in the store and when the source is notified of message stability. If using a stability ACK-based flight size on a persistent 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)
repository-allow-receiver-paced-persistence	Specifies if the repository allows receiver-paced persistence (1). If allowed, the source may optionally request RPP with **ume_receiver_paced_persistence (source)**. Both must be done for RPP to be in effect.	0 (store does not allow the source to specify RPP)
repository-allow-ack-on-reception	For RPP, specifies if the repository allows the repository to perform "ack on reception" (1). If allowed, the source may optionally request "ack on reception" with **ume_repository_ack_on_reception (source)**. Both must be done for "ack on reception" to be in effect. For SPP and memory stores, this option has no effect. See RPP Normal Operation for more information.	0 (store does not allow the source to specify ack-on-reception behavior)

Option	Description	Default Value
repository-disk-write-delay	For topics with a repository-type of disk or reduced-fd, specifies the maximum delay in milliseconds after message reception before the repository persists a message to disk. For RPP, the source may optionally override this value with a value less than or equal with the source configuration option **ume_write_delay (source)**. (units: milliseconds)	0 milliseconds
source-flight-size-bytes-maximum	With RPP, specifies the maximum number of in-flight payload bytes that the source is allowed to configure with **ume_flight_size_bytes (source)**. If the source attempts to configure **ume_flight_size_bytes (source)** greater than the store's source-flight-size-bytes-maximum, the source registration is rejected. For SPP stores, this option has no effect (i.e. the source is not restricted in its configuration). See Persistence Flight Size for more information. (units: bytes)	4194304 bytes (4MB)

11.3 Option Types for ume-attributes Elements

As mentioned in Options for a Store's ume-attributes Element, all options configured for '<ume-attributes>' require an '<option>' type, which specifies the scope of the option.

For example, here is a store configuration which illustrates several options being set of varying types:

```
<?xml version="1.0"?>
  <stores>
    <store name="test-store" port="14567">
      <ume-attributes>
        <option type="store" name="disk-cache-directory" value="cache"/>
            <option type="lbm-context" name="transport_lbtrm_rate_interval"
                value="100">
      </ume-attributes>
      <topics>
        <topic pattern="test.*" type="PCRE">
          <ume-attributes>
            <option type="lbm-receiver" name="transport_lbtrm_send_naks" value="0">
            <option type="lbm-source" name="transport" value="lbtru">
            <option type="store" name="repository-size-limit" value="209715200"/>
          </ume-attributes>
        </topic>
      </topics>
    </store>
  </stores>
```

The following table describes the Option Types:

Option Type	Description	Default Value
lbm-receiver	Allows you to configure receiver-scope options that you usually specify in an Ultra Messaging configuration file or set using lbm_∗_attr_setopt(). 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"> '<option>' is a child of '<ume-attributes>', but you can use option type lbm-receiver within only a '<topic>' element, not a '<store>' element.	None - this is a required attribute.
lbm-context	Allows you to configure context-scope options that you usually specify in an Ultra Messaging configuration file or set using lbm_∗_attr↩ _setopt(). 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"> '<option>' is a child of '<ume-attributes>', but you can use option type lbm-receiver within only a '<store>' element, not a '<topic>' element.	None - this is a required attribute.
lbm-source	Allows you to configure source-scope options that you usually specify in an Ultra Messaging configuration file or set using lbm_∗_attr_↩ setopt(). For example, you could change the transport by including the following within the '<ume-attributes>' element: <option type="lbm-source" name="transport" value="lbtru"> '<option>' is a child of '<ume-attributes>', but you can use option type lbm-source within only a '<topic>' element, not a '<store>' element.	None - this is a required attribute.
store	Option type used for all ume-attributes configured for the store element and its topic element.	None - this is a required attribute.

11.4 umestored Configuration DTD

The DTD for UM Store configuration has evolved over time:

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
1.3	November 2016	UM 6.10	Addition of '<xml-config>' element (under '<daemon>').

Here is the current version:

```
<!ELEMENT ume-store (daemon, stores?)>
<!ATTLIST ume-store version CDATA #REQUIRED>
<!ELEMENT daemon (log | uid | pidfile | gid | lbm-config | xml-config |
    lbm-license-file | web-monitor | daemon-monitor)*>
<!ELEMENT log ( #PCDATA )>
<!ATTLIST log type (file | console) "console">
<!ATTLIST log frequency (disable | daily | hourly | test) "disable">
<!ATTLIST log size 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 xml-config ( #PCDATA )>
<!ATTLIST xml-config xml:space (default | preserve) "default">
<!ATTLIST xml-config application-name CDATA #IMPLIED>
<!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">
<!ELEMENT stores (store*)>
<!ELEMENT store (publishing-interval | 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*)>
<!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) #IMPLIED>
<!ATTLIST option name CDATA #REQUIRED>
<!ATTLIST option value CDATA #REQUIRED>
<!ELEMENT daemon-monitor (lbm-config | publishing-interval |
    remote-snapshot-request | remote-config-changes-request)*>
<!ATTLIST daemon-monitor topic CDATA "umestore.monitor">
<!ELEMENT publishing-interval (group+)>
<!ELEMENT group EMPTY>
<!ATTLIST group name (default | store | source | receiver | disk | config | memory)
    #REQUIRED>
```

```
<!ATTLIST group ivl CDATA #REQUIRED>
<!ELEMENT remote-snapshot-request EMPTY>
<!ATTLIST remote-snapshot-request allow (0 | 1) "0">
<!ELEMENT remote-config-changes-request EMPTY>
<!ATTLIST remote-config-changes-request allow (0 | 1) "0">
```

11.5 Store Configuration Example

Store daemon with one store.

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <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>
```

11.5.1 xml-config Tag

The '<xml-config>' tag is used to load a UM configuration file which is used by the store as it creates UM objects (contexts, receivers, etc.). This example applies the configuration specified in the "tnwgd" application tag of the file "lbm_cfg.xml":

```
<ume-store version="1.3">
    <daemon>
        ...
```

```
        <xml-config application-name="tnwgd">lbm_cfg.xml</lbm-xmlconfig>
    </daemon>
```

If an '`application-name`' attribute is not supplied, "umestored" is assumed. If it is desired to load the unnamed configuration, use an empty application name as follows:

```
<ume-store version="1.3">
    <daemon>
        ...
        <xml-config application-name="">lbm_cfg.xml</lbm-xmlconfig>
    </daemon>
```

Chapter 12

Store Daemon Statistics

This section contains details on the Store's Daemon Statistics feature. **You should already be familiar with the general information contained in Daemon Statistics.**

12.1 Store Daemon Statistics Structures

The different message types are:

- **LBM_UMESTORE_DMON_MPG_SMART_HEAP_STATS**
- **LBM_UMESTORE_DMON_MPG_STORE_STATS**
- **LBM_UMESTORE_DMON_MPG_REPO_STATS**
- **LBM_UMESTORE_DMON_MPG_DISK_STATS**
- **LBM_UMESTORE_DMON_MPG_RCV_STATS**
- **LBM_UMESTORE_DMON_MPG_STORE_CONFIG**
- **LBM_UMESTORE_DMON_MPG_STORE_PATTERN_CONFIG**
- **LBM_UMESTORE_DMON_MPG_STORE_TOPIC_CONFIG**
- **LBM_UMESTORE_DMON_MPG_REPO_CONFIG**
- **LBM_UMESTORE_DMON_MPG_RCV_CONFIG**

Each one has a specific structure associated with it, as detailed in **umedmonmsgs.h**.

Note that message types ending with "_CONFIG" are in the config category, while message types ending with "↩ _STATS" are in the stats category. See **Daemon Statistics Structures** for information on how the two categories are handled differently.

12.1.1 Store Daemon Statistics Byte Swapping

A monitoring application receiving these messages must detect if there is an endian mismatch (see **Daemon Statistics Binary Data**). The header structure **umestore_dmon_msg_hdr_t** contains a 16-bit field named `magic` which

is set equal to **LBM_UMESTORE_DMON_MAGIC**. The receiving application should compare it to **LBM_UME↩ STORE_DMON_MAGIC** and **LBM_UMESTORE_DMON_ANTIMAGIC**. Anything else would represent a serious problem.

If the receiving app sees:

```
magic == LBM_UMESTORE_DMON_MAGIC
```

then it can simply access the binary fields directly. However, if it sees:

```
magic == LBM_UMESTORE_DMON_ANTIMAGIC
```

then *most* (but not all) binary fields need to be byte-swapped. See umedmon.c for an example, paying special attention to the macros COND_SWAPxx (which *conditionally* swaps based on the magic test) and the functions byte_swapXX() (which performs the byte swapping).

However, there are some binary fields which must never be swapped, regardless of the endian. This is indicated in the documentation. For example, **umestore_store_dmon_config_msg_t_stct::store_iface** says "NOTE: This field should NOT be byte-swapped." Here's how that field might be accessed:

```
in.s_addr = msg->store_iface;
printf("Store IP address / port: %s / %d\n",
    inet_ntoa(in), COND_SWAP16(msg_swap, msg->store_port));
```

As you can see, store_iface is not byte swapped, but store_port (conditionally) is swapped.

12.1.2 Store Daemon Statistics String Buffers

There are some messages which contain string buffers at the ends of the messages. Strings in these data structures are always null-terminated. Be aware that these messages are not sent as fixed-length equal to the size of the data structure, but rather are sent with only the bytes required by the string (including the final null). For example, the structure **umestore_store_pattern_dmon_config_msg_t** contains the field **umestore_store_pattern_dmon_↩ config_msg_t_stct::pattern_buffer** which is char array of size LBM_UMESTORE_DMON_TOPIC_PATTERN↩ _STRLEN. If pattern_buffer is set to ".*", then only 3 bytes (including the null string terminator) are sent for that field.

Contrast this with **UM Router Daemon Statistics String Buffers**.

This becomes more complicated when there are multiple strings in one message. For example, consider **umestore_store_dmon_config_msg_t**. This message contains three strings: store name, cache directory name, and state directory name. But a single char array is declared:

```
char string_buffer[LBM_UMESTORE_DMON_STORE_NAME_STRLEN + (2 *
    LBM_UMESTORE_DMON_FILENAME_MAX_STRLEN)
```

The three strings are packed into that buffer, only taking up as much space as is necessary. I.e. if the three strings are "a", "b", and "c", only 6 bytes of the buffer will be consumed (each string has a null).

To make it easier for the code to find the three strings, the structure has three offset variables: store_name↩ _offset, disk_cache_dir_offset, and disk_state_dir_offset. These are byte offsets from the start of the entire structure. So, to access the store name, the monitoring application might use:

```
umestore_store_dmon_config_msg_t *store_config_msg = ... /* ptr to incoming msg */
char *state_dir_name = (char *)store_config_msg +
                       store_config_msg->store_name_offset;
```

(The practice of using offsets from the start of the structure allows for greater flexibility in ensuring inter-version compatibility.)

12.1.3 Store Daemon Statistics Retx Counts

There is a set of fields in **umestore_store_dmon_stat_msg_t** which give statistics on recovery operations initiated by receivers:

- **umestore_store_dmon_stat_msg_t_stct::ume_retx_req_rcv_count**
- **umestore_store_dmon_stat_msg_t_stct::ume_retx_req_serviced_count**
- **umestore_store_dmon_stat_msg_t_stct::ume_retx_req_drop_count**

The web monitor's Store Web Monitor Store Page has a manual function labeled Reset Rate Stats which clears those "ume_retx_..._count" fields. This is a useful function for users who use the web monitor as their primary monitoring tool, but for users who depend on the published Daemon Statistics, it can be disruptive for the counts to be cleared on-demand.

The field **umestore_store_dmon_stat_msg_t_stct::ume_retx_stat_interval** contains the seconds since the last Reset Rate Stats operation. If the user has not used Reset Rate Stats, then ume_retx_stat_interval contains the seconds since the store's startup.

12.2 Store Daemon Statistics Configuration

There are two places in the Store configuration file that Daemon Statistics are configured:

- The Daemon-monitor Element inside the <daemon> definition. Configures all aspects of the Store Daemon Statistics feature, including publishing intervals.
- The Publishing-interval Element inside a <store> definition. Configures only the publishing intervals on a store basis.

Here is an example of configuring daemon statistics.

```
<ume-store version="1.3">
<daemon>
  <daemon-monitor topic="bozo">
    ...
    <publishing-interval>
      <group name="default" ivl="3"/>
      <group name="config" ivl="120"/>
    </publishing-interval>
    <remote-snapshot-request allow="1"/>
    <remote-config-changes-request allow="1"/>
  </daemon-monitor>
<daemon>
<stores>
  <store name="store0" port="12000">
    <publishing-interval>
      <group name="default" ivl="6"/>
      <group name="config" ivl="120"/>
    </publishing-interval>
    ...
  </store>
  <store name="store1" port="12001">
    ...
  </store>
</stores>
```

In this example, all stats-type messages are (conditionally) published on a 3-second interval, except those of store0, which are published (conditionally) on a 6-second interval. All config-type messages are published (unconditionally) on a 120-second interval.

12.3 Store Daemon Statistics Requests

The Store Daemon supports a monitoring application to send a specific set of requests to control the operation of Daemon Statistics. The remote-snapshot-request and remote-config-changes-request configuration elements control whether the Store enables this request feature (defaults to disabled).

If enabled, the monitoring application can send a command message to the store in the form of a topicless unicast immediate "request" message (see **lbm_unicast_immediate_request()** with NULL for topic). The format of the message is a simple ascii string, with or without null termination. Due to the simple format of the message, no data structure is defined for it.

When the Store receives and validates the command, it sends a UM response message back to the requesting application containing a status message (which is *not* null-terminated). If the status was OK, the Store also performs the requested action.

The example program `umedcmd.c` demonstrates the correct way to send the messages and receive the responses.

Commands enabled by remote-snapshot-request:

version

The Store returns in its command response the value of **LBM_UMESTORE_DMON_VERSION**. No daemon statistics messages are published.

snap memory

The Store immediately publishes the memory usage message **LBM_UMESTORE_DMON_MPG_SMART_HEAP_STATS**.

snap src

The Store immediately publishes the source repository statistics message(s) **LBM_UMESTORE_DMON_MPG_REPO_STATS**.

snap rcv

The Store immediately publishes the receiver statistics message(s) **LBM_UMESTORE_DMON_MPG_RCV_STATS**.

snap disk

The Store immediately publishes the disk statistics message(s) **LBM_UMESTORE_DMON_MPG_DISK_STATS**.

snap store

The Store immediately publishes the store statistics message(s) **LBM_UMESTORE_DMON_MPG_STORE_STATS**.

snap config

The Store immediately publishes the store config category messages **LBM_UMESTORE_DMON_MPG_STORE_CONFIG**, **LBM_UMESTORE_DMON_MPG_STORE_PATTERN_CONFIG**, **LBM_UMESTORE_DMON_MPG_STORE_TOPIC_CONFIG**, **LBM_UMESTORE_DMON_MPG_REPO_CONFIG**, and **LBM_UMESTORE_DMON_MPG_RCV_CONFIG**

Commands enabled by remote-config-changes-request:

memory N

Set the publishing interval for memory usage.
For example: `memory 5`

src N

Set the publishing interval for source repository statistics messages. This command can be preceded by a store name in double quote marks to only set the publishing interval for that store.
For example: `"store1" src 5`

rcv N

Set the publishing interval for receiver statistics messages. This command can be preceded by a store name in double quote marks to only set the publishing interval for that store.
For example: `"store1" rcv 5`

disk N

Set the publishing interval for disk statistics messages. This command can be preceded by a store name in double quote marks to only set the publishing interval for that store.
For example: `"store1" disk 5`

store N

Set the publishing interval for store statistics messages. This command can be preceded by a store name in double quote marks to only set the publishing interval for that store.
For example: `"store1" store 5`

config N

Set the publishing interval for config category messages. This command can be preceded by a store name in double quote marks to only set the publishing interval for that store.
For example: `"store1" config 5`

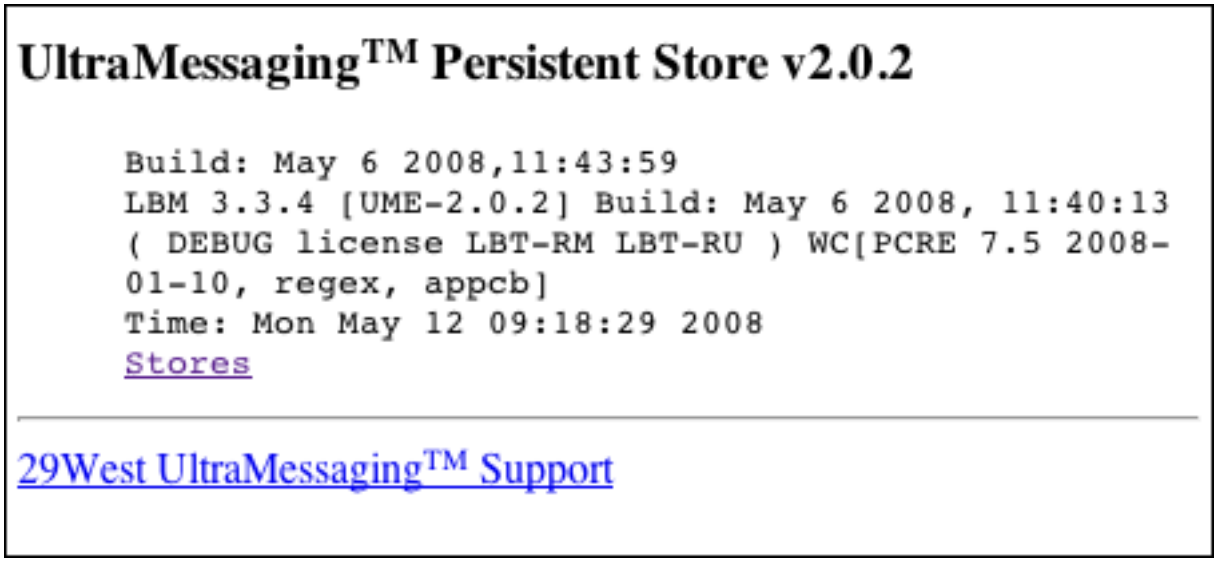
Chapter 13

Store 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. 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.

13.1 Store Web Monitor Index Page

Here is an image of the Web Monitor's *Index* (main) page:



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

The "Stores" link displays the Store Web Monitor Stores Page.

13.2 Store Web Monitor Stores Page

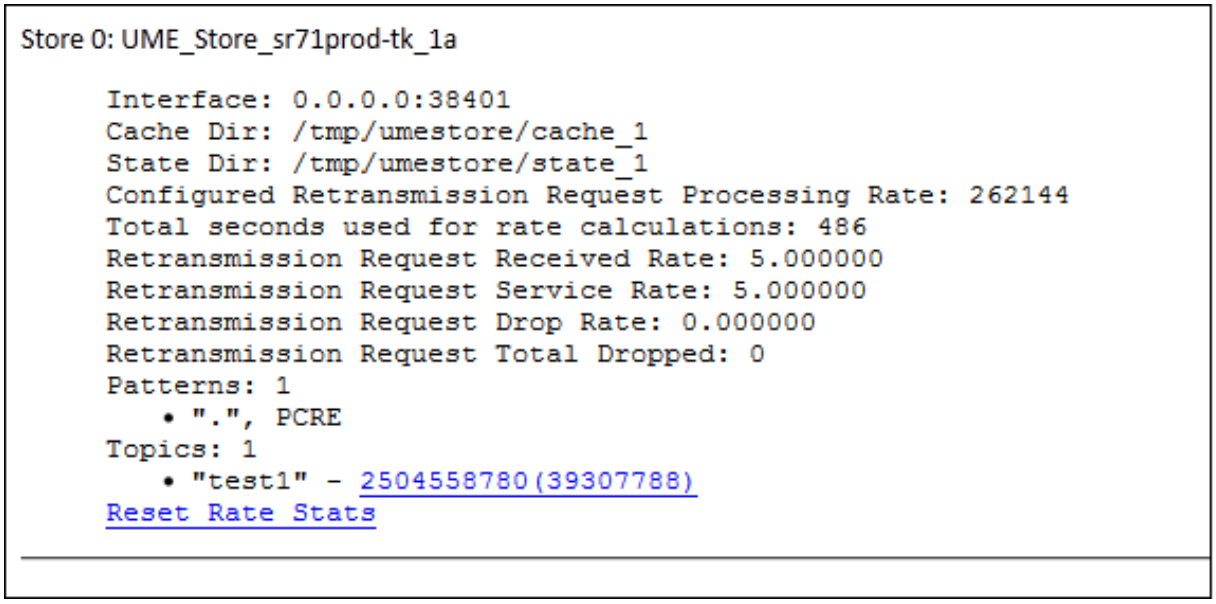
Here is an image of the Web Monitor's *Stores* page:

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".

Each store name is a clickable link, which displays the Store Web Monitor Store Page for that store.

13.3 Store Web Monitor Store Page

Here is an image of the Web Monitor's *Store* page:



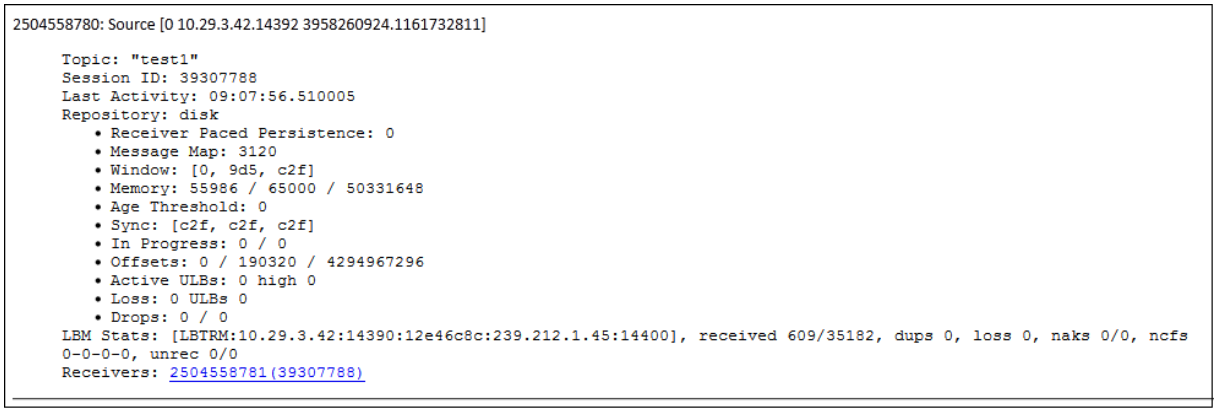
This page shows the following information about the store.

Item	Description
Interface	This store is listening on all interfaces (0.0.0.0) on port 38401.
Cache Dir	Pathname for disk store message cache directory. This would be configured as a store attribute in the store's XML configuration file. For example: <option type="store" name="disk-cache-directory" value="cache/" />

Item	Description
State Dir	Pathname for disk store state directory. This would be configured as a store attribute in the store's XML configuration file. For example: <option type="store" name="disk-state-directory" value="state/" />
Configured Retransmission Request Processing Rate	Current value for the store's retransmission-request-processing-rate option setting.
Total Seconds Used for Rate Calculations	Accumulating counter that displays the number of seconds since the last rate reset. The Web Monitor divides the Retransmission Request Received, Retransmission Request Service and Retransmission Request Drop totals by the Total Seconds to calculate the rates displayed. If you click the Reset Rate Stats, the Web Monitor resets this value to zero.
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. For example: <topic pattern="test.*" type="PCRE">
Topics	Displays the topic names and Registration ID (Session ID) for any sources publishing on the topic. The screen examples display one topic, test1 - 2504558780(39307788). Each Registration ID (Session ID) is a clickable link, which displays the Store Web Monitor Source Page for that source.
Reset Rate Stats	Click the Reset Rate Stats link to reset the retransmission rates. After clicking the link, The Web Monitor rests Total Seconds Used for Rate Calculations to zero and displays a page with the store number and the message, 'Rate Statistics have been reset'.

13.4 Store Web Monitor Source Page

Here is an image of the Web Monitor's *Source* page:



The first line in the page contains is interpreted as follows:

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 (context)**.
3958260924	The source's transport session index.
1161732811	The source's topic index within the transport session, 3958260924.

The remaining fields are described in the following table:

Source Page Item	Description
Topic	test is the source's topic string.
Session ID	39307788 is the source's Session ID.
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. Possible values are memory, disk or reduced-fd.
Receiver Paced Persistence	Setting for Receiver-paced Persistence (RPP), which is a repository option both the repository and source must enable. A value of 0 means RPP is not enabled and the repository is using the default Source-paced persistence. A value of 1 means RPP is enabled.
Message Map: 3120	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.

Source Page Item	Description
Window: [0, 9d5, c2f]	Window format is: trail_sqn, mem_trail_sqn, lead_sqn • trail_sqn, 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. The trailing sequence number changes if the store reaches a disk file size limit and then deletes the oldest messages. • mem_trail_sqn, 9d5, is the trailing sequence number for messages in memory. 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. • lead_sqn, c2f, is the leading sequence number, which is the newest 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.
Memory: 55986 / 65000 / 50331648	Memory format is: repository memory size / repository size threshold / repository size limit • repository memory size, 55986, is the number of bytes of messages in memory, which includes headers and store overhead. • repository size threshold, 65000, is the repository-size-threshold topic option found in the store's XML configuration file. • repository size limit, 50331648, is the store's repository-size-limit topic option found in the store's XML configuration file. 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	Age Threshold, 0, is the store's repository-age-threshold topic option found in the store's XML configuration file.

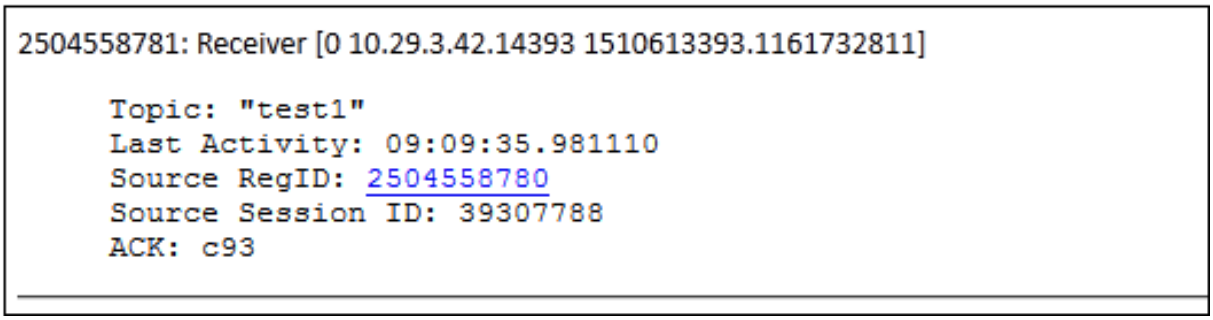
Source Page Item	Description
Sync: [c2f, c2f, c2f]	Pertains to disk or reduced-fd repositories only. Sync format is: sync_↩ complete_sqn, sync_sqn, contig_sqn • sync_complete_sqn, c2f, Most recent sequence number that the Operating System has confirmed persisting to disk. • sync_sqn, c2f, Most recent 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. • 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.

Source Page Item	Description
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. • 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. 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.

Source Page Item	Description
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.
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 and accompanying Session ID for the receivers listening on the source's topic. Click on the receiver Registration ID (Session ID) to display the Store Web Monitor Receiver Page to review information about the receivers for that persisted topic.

13.5 Store Web Monitor Receiver Page

Here is an image of the Web Monitor's *Receiver* page:



The first line in the page contains is interpreted as follows:

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 (context)**.
1510613393	The receiver's transport session index.
1161732811	The source's topic index within the transport session, 1510613393.

The remaining fields are described in the following table:

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 Store Web Monitor 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.