Ultra Messaging (Version 6.17)

Guide for Persistence

Contents

Chapter 1

Introduction

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

For policies and procedures related to Ultra Messaging Technical Support, see `UM Support`.

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

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.

Persistence

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 keeping them for a period of time so that receivers can recover messages if necessary.

A "Store Process" contains one or more independent "Store instances", where an "instance" is an independent, addressable, and configurable component. See Store Processes and Instances for more detail.

Store instances are used by persistent subscribers to recover messages that cannot be recovered from the source by the transport protocol. For example, messages published prior to the subscriber joining the transport can be recovered from a Store instance. After a period of overload or network outage that leads to loss of live messages, the subscriber uses a Store instance to recover messages that could not be recovered by the transport protocol.

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 Store instances to run independently of sources and receivers and in parallel with messaging. The Store does not interfere with message delivery to receiving applications.

This document is oriented mostly to programmers. See also the Operations Guide chapters **Running Persistent Stores (umestored)**, **Persistent Store Crashed**, **Persistent Sending Problems**, and **UM Persistent Store Log Messages**.

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.

The user can choose between two different persistence modes:

- Source-paced Persistence (SPP) - default mode - the rate of message consumption by receivers does not constrain the rate a source can send. The Store instance writes all messages to 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. The Store instance only writes message to non-volatile storage if one or more required RPP receivers is absent or slow in consuming the messages. Messages are deleted from the Store instance once all receivers have consumed the RPP message. See RPP: Receiver-Paced Persistence.

3.1 Persistent Store Concept

UM uses a daemon program known as the Store to persist source (publisher) and receiver (subscriber) state. A Store instance can persist state in memory as well as on disk. State is persisted on a per-topic, per-source basis by the Store. Along with each publisher's state is a message cache containing the full message contents of recently-sent messages by the source.

The purpose of the Store is to allow receivers to recover messages that the receiver was not able to get directly from the source.

The Store is an independent component, not part of the source. If a persistent publisher fails, that source's messages are maintained by the Store according to configurable retention policies.

Note that the design of UM's persistence allows a maximum of 2,147,483,647 messages ($2**31$ - 1) to be persisted.

Stores can be configured to be disk-based or memory-only. A disk-based Store uses memory as temporary storage while messages are written to disk. Memory-only Stores only hold messages in memory. The memory-only Stores have higher throughput, while disk-based Stores have greater message capacity.

Note that most UM deployments only use disk-based Stores. Most of this document is written with that assumption.

3.2 Persistence Configuration Concept

It is important to remember the different kinds of configuration.

- Applications create UM objects (contexts, sources, receivers) using the UM library. Those objects must be configured to control their operation and behavior using **"LBM configuration options"**. An application typically uses an **"LBM configuration file"** in either XML or flat format. For full details on LBM configuration options, see UM Configuration Guide

- A Store Process is configured using a **"Store configuration file"** in XML format. For full details on Store configuration files, see Configuration Reference for Umestored.

- A Store Process also internally creates UM objects (contexts, sources, receivers) using the UM library. The Store's objects must also be configured using one or more LBM configuration files.

So Stores need two kinds of configuration files: Store configuration files and LBM configuration files. Applications only need LBM configuration files.

3.3 Registration Identifier Concept

UM persistence identifies sources and receivers with Registration Identifiers, also called Registration IDs or RegI↩
Ds. A RegID is a 32-bit number that uniquely identifies a source or a receiver to a Store instance. This means that RegIDs are also specific to a Store instance and can be reused between individual Store instances, 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.

While RegIDs can be managed directly by applications, Informatica recommends the use of Session IDs instead. See Managing RegIDs with Session IDs.

3.4 Delivery Confirmation Concept

A persistent receiver provides confirmation (acknowledgement) to the Store instance 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 LBM 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 LBM 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

3.5 Retention Policy

Sources and Persistent Stores retain messages in memory according to a set of rules collectively called the retention policy. The rules specify when UM will remove a message from memory, an action called "reclaiming" (because the memory is reclaimed from the buffer). Note that reclaiming a message from memory does not mean the message can no longer be recovered. The opposite is true - a message is reclaimed from memory only after it is stable on the Stores.

A message must satisfy every rule before it can be reclaimed. Conversely, any message not complying with all rules will not be reclaimed. A source or Store instance retains messages in memory 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.

3.6 Message Stability Concept

Sources send messages to both receivers and to Store instances. 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.

3.7 Quorum/Consensus Store Failover

Typically, multiple Store instances are deployed as a group for 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 Store instances are configured, messaging can continue as long as at least 2 are operational. If 5 Store instances are configured, messaging can continue if at least 3 are operational. (Quorum/Consensus requires an odd number of Store instances in the QC group.)

Sources define the QC group by the LBM configuration option **ume_store (source)**, one for each Store in the group.

Chapter 4

Persistence Architecture

As shown in the diagram, UM provides messaging functionality as well as persistent operation.
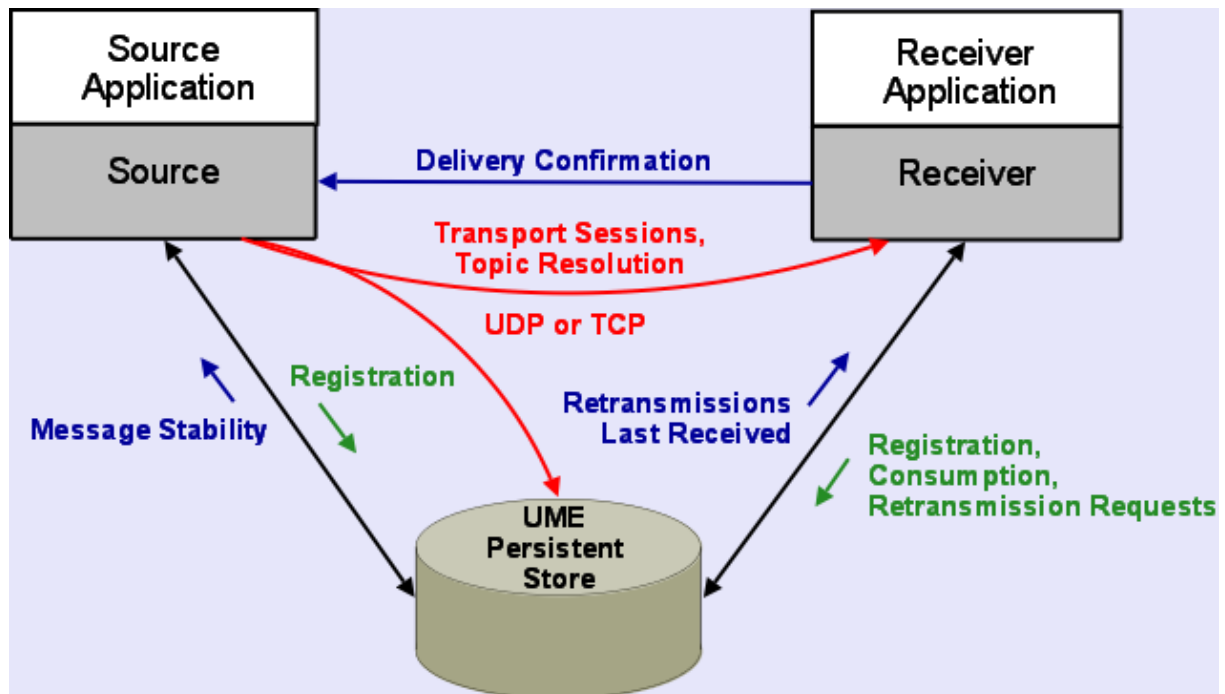
The highlights of this architecture are:

- Sources communicate with Store instances

- Receivers communicate with Store instances

- Sources communicate with receivers

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

4.1   Persistent Store Architecture

The `umestored` program (the final "d" stands for "daemon") runs the Store Process. You can configure multiple Store instances per Store Process using the UMP Element "<store>" in the Store configuration file. See Configuration Reference for Umestored. Individual Store instances 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". Each Store Process has an optional Web Monitor for statistics monitoring. See Store Web Monitor.
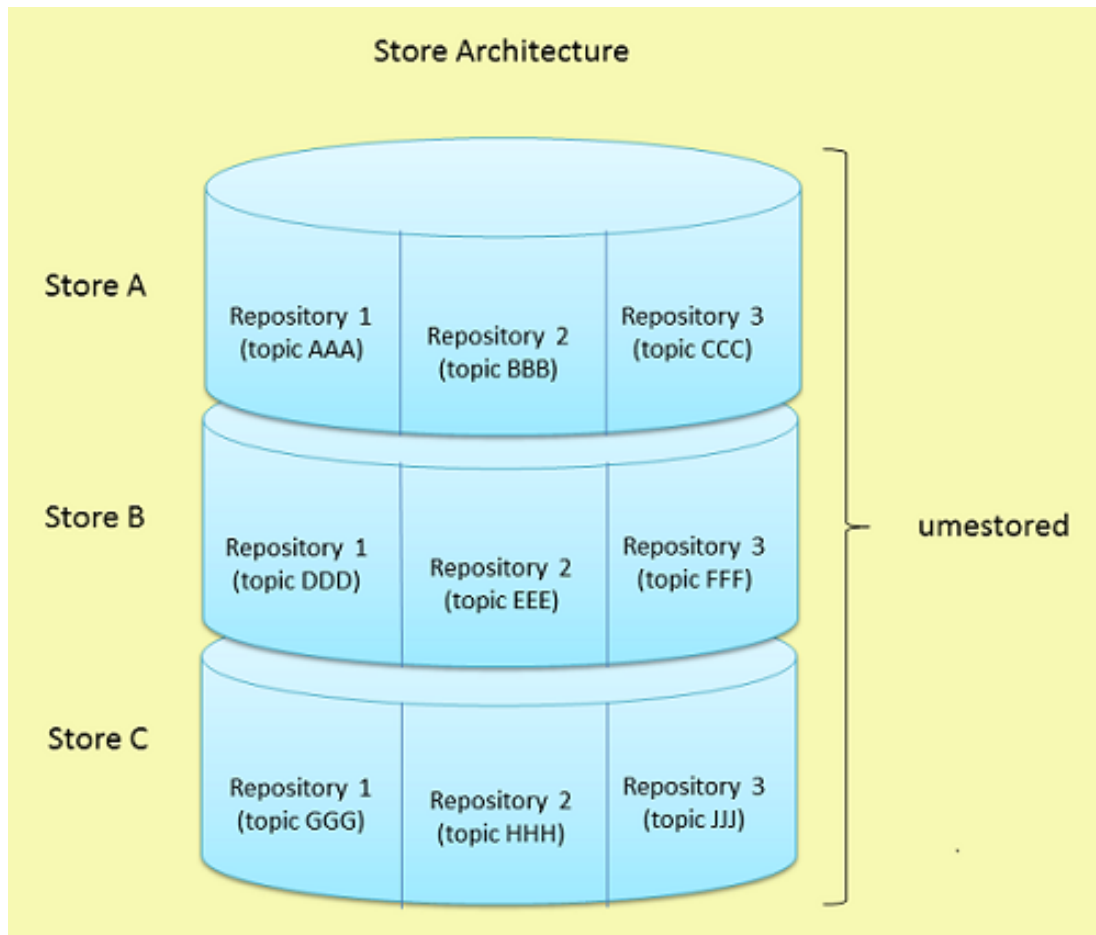
4.1.1   Store Processes and Instances

When the Store Process is started on a host, the process is known as the "Store Process". That Store Process contains one or more "Store instances". A Store instance is an independent, addressable, and configurable component. Each Store instance is implemented with a set of interacting program threads. The threads of one Instance do not interact or contend with the threads of other Instances in the same Process.

There is very little difference between running one Store Process with two Store instances compared to two Store Processes with one Store instance each. They function and perform mostly the same. The reasons for choosing one over the other have mostly to do with operational convenience. For example, running fewer processes on a host is sometimes easier to manage. So operational simplicity suggests combining multiple Store instances into a single Store Process.

On the other hand, there are times when it is desired to shut down a Store instance. But Store instances cannot be shut down individually; an entire Store Process must be shut down. For example: as message rates increase, you may find that the host's CPU consumption is getting too high. You might want to migrate half of the Store instances

to a different host. But if all your Store instances are in one Store Process, it is more disruptive perform the migration since it requires shutting down the entire process and re-configuring. So operational flexibility suggests assigning each desired Store instance to its own Store Process.

One specific case where a single Store Process with multiple Store Instances is generally preferred: using the Store Process as a Windows Service. There is no simple way to run multiple copies of the Store Windows Service.

4.1.2 Source Repositories

Within a Store instance, you configure repositories for individual topics, and each can have its own set of <topic> options that affect the repository's type, size, liveness behavior, among other options. If you have multiple sources sending on the same topic, the Store instance creates a separate repository for each source. UM uses the repository options configured for the topic to apply to each source's repository.

For example, if you specify 48 MB for the size of the repository and have 10 sources sending on the topic, the Store instance requires 480 MB of storage for that topic.

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

- memory - the repository maintain both state and data only in memory, not disk.

- disk - the repository maintains state and data on disk, but also uses a memory cache.

There is also a repository type called "no-cache", which is deprecated and will be removed in a future UM version. The "no-cache" repository maintains state (last sequence numbers published and consumed) but does not maintain message content. It is deprecated due to lack of compelling use cases.

Note that the Store instances within a Store Process can have different repository types.

4.1.3 Repository Thresholds and Limits

The Store is designed to retain messages in case they are needed for future recovery. Of course, it is not possible to extend this retention to infinity, so the Store must be configured with policies regarding the removal of messages. There are three possible policies that can be established:

1. Repository size limit (required)

2. Message age limit (optional, for source-paced persistence).

3. Consumption by all required receivers (for receiver-paced persistence).

In all cases, the repository's size is limited to prevent exhaustion of storage. With source-paced persistence, when the repository size limit is reached, the oldest messages are overwritten by new messages. With receiver-paced persistence, when the repository size limit is reached, the source is prevented from sending more messages.

For receiver-paced persistence, when all required receivers acknowledge consumption of a message, it is removed from the Store. But note that if the required receivers to not acknowledge consumption of messages and the repository fills before the oldest messages are acknowledged, the repository size is enforced and the source is blocked from sending more messages.

Note

> When you configure a size limit, it applies to a single persisted source. For example, if you have two publishing applications and each one sends to the topic "EventStream", two separate repositories will be created, one for each source, and each repository will be allowed to grow to the configured size limit. You must provision your repository sizes with knowledge of how many persisted sources will be serviced by a given Store.

The size configuration options differ depending on whether you are implementing a Memory Repository Store (repository-type **"memory"**) or a Disk Repository Store (repository-type **"disk"**).

Memory Repository Size

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

Note that the design of UM's persistence allows a maximum of 2,147,483,647 messages ($2**31 - 1$) to be persisted.

- repository-age-threshold - This value determines how long the memory repository retains messages. Messages in memory that exceed this time can be deleted from the memory cache.

- 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. Do not specify a limit that would allow more than 2,147,483,647 messages to be stored.

Disk Repository Size

A disk 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.

Note that the design of UM's persistence allows a maximum of 2,147,483,647 messages ($2**31 - 1$) to be persisted.

The Store has four configuration options that manage its size relative to its capacity.

- 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. Do not specify a limit that would allow more than 2,147,483,647 messages to be stored.

- 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. Do not specify a limit that would allow more than 2,147,483,647 messages to be stored.

4.1.4   Persistent Store Fault Tolerance

Sources and receivers register with a Store instance and use individual repositories within the Store. Sources can use redundant repositories configured in multiple Stores Instances in Quorum/Consensus (QC) arrangement for fault tolerance. Be aware that the arrangement of Store instances into Quorum/Consensus groups is a function of the source. I.e. the individual Stores of a QC group are not aware of each other and do not coordinate their activities.

Informatica strongly recommends that the Store instances of a QC group run on separate physical hosts.

4.1.5   Identifying Persistent Stores

A persistent source must be configured to identify one or more Stores to provide persistence services. The source configuration can identify Store instances with one of:

- IP:port or domainID:IP:port using **ume_store (source)**, see Store Address

- Store's context name using **ume_store_name (source)**, see Named Stores).

In either case, the store should be told which interface to bind to, which defines its IP address. This is done with the UMP Element "<store>" in the Store's configuration file. There is a shortcut available to simplify the Store configuration file; see Using a CIDR Range of IP Addresses.

Store Address: Identify Store with IP:Port

Using IP:port is feasible in deployments where there is no DRO; i.e. all components are in a single **Topic Resolution Domain** (TRD). Deployments that include DRO and have multiple TRDs require that the domain ID be added to the address: domainID:IP:port.

Configure Store instance for a single IP:port.

1. Identify the Store with only the IP:port, specified with the UMP Element "<store>" in the Store's configuration file. For example:

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

2. Configure the source with the IP:port using the LBM configuration option **ume_store (source)** so sources can find and register with the Store instance.

   ```
   source ume_store 10.29.3.16:14567
   ```

Named Stores: Identify Store with Context Name

A Store is configured with a context name. Sources are then configured to specify the Stores by their names, instead of their IP:Port.

1. Give the Store's context a name using the context-name option in the Store configuration file. For example:

   ```
   <store name="newyork-1" port="14567" interface="10.29.3.0/24">
   <ume-attributes>
      <option type="store" name="context-name" value="NEWYORK-1"/>
   </ume-attributes>
   ```

2. Configure the source with the name of the Store's context using the LBM configuration option **ume_store_↩ name (source)** so sources can find and register with the Store.

   ```
   source ume_store_name NEWYORK-1
   ```

   Note that you did not have to determine the full IP address of the store's host.

Store context names can be used with or without DROs. UM automatically resolves and maintains a mapping between a Store's context name and its domain ID, IP address and port, as follows:

- Store advertises its context name at startup and in response to queries from sources.

- If a Store receives a context name advertisement that matches its own context name, that Store issues a warning in the Store's log. This represents an invalid configuration and can produce unpredictable results. Always ensure that Store context names are unique within a UM deployment.

- Sources using Store context names issue an information message to the application every time a resolved context name changes its DomainID:IPaddress:port.

Shortcut: Using a CIDR Range of IP Addresses

Configure a Store with a CIDR range of IP addresses (see **Specifying Interfaces**). This allows multiple Store daemon instances which only differ by their IP address to be configured the same. At initialization time, each Store daemon instance will determine its IP address using the CIDR specification. However, be aware that sources will need to use the full IP address.

1. Identify the Store with a range of IP addresses specified in the Store configuration file. For example:

   ```
   <store name="newyork-1" port="14567" interface="10.29.3.0/24">`
   ```

   When the Store Process initializes, UM will choose a network interface within that IP address range (10.29.↩
   3.0 - 10.29.3.255).

2. Configure the source with the IP:port using the LBM configuration option **ume_store (source)** so sources can find and register with the Store. **You must specify the full IP address, not the CIDR range.**

   ```
   source ume_store 10.29.3.16:14567
   ```

3. Alternatively, you can use the Named Stores feature so that the source doesn't need to specify the full IP.

Chapter 5

Operational View

Sources, receivers, and Store instances 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.

This document is oriented mostly to programmers. See also the Operations Guide chapters **Running Persistent Stores (umestored)**, **Persistent Store Crashed**, **Persistent Sending Problems**, and **UM Persistent Store Log Messages**.

Note

> If your application is running with the LBM configuration option **request_tcp_bind_request_port (context)** set to zero, UIM port binding (also known as "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 Store instances 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 instance:

1. Source advertises topic over topic resolution transport

2. (optional) Source queries for and resolves Store context names

3. Source registers with Store by unicast

4. Source sends SRI over configured transport

The following diagram illustrates network flow during the registration process.

Source Registration
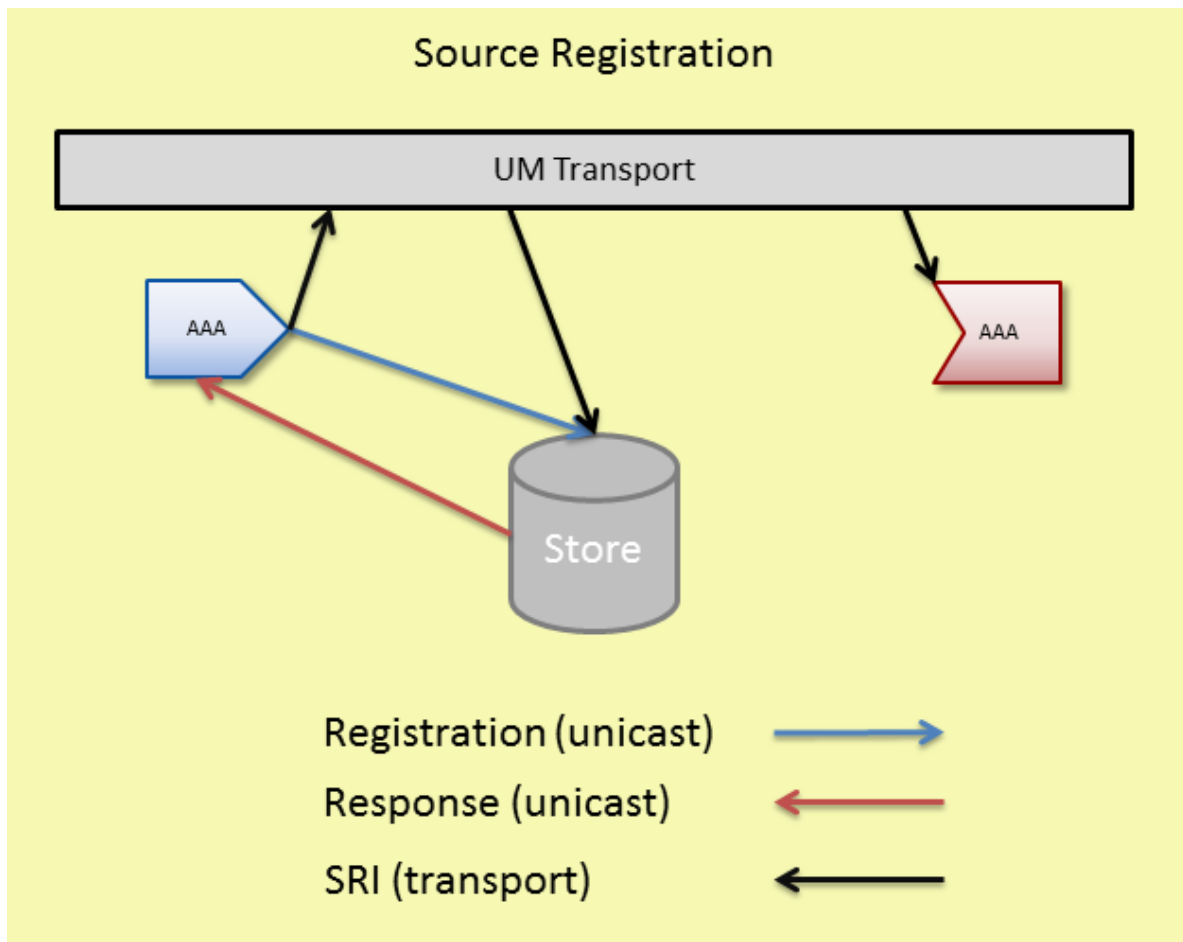
Sources can find the correct Store instance(s) to register with from the values configured for it in the LBM configuration options **ume_store (source)** or **ume_store_name (source)**. The LBM configuration option **ume_store (source)** contains the IP address, TCP port, registration ID, and group index for the Store(s) to be used by the source. The LBM 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 context name is configured with the context-name option in the Store configuration file. See Identifying Persistent Stores and the UMP Element "<store>".

Sources unicast registrations to the Store instance. 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 required Stores, the source delivers a Registration Complete event to the publisher and sends an SRI over the source's **transport session**. For multiple Stores in the QC group, the source determines the required number of Stores based on the LBM configuration option **ume_retention_↩ intragroup_stability_behavior (source)**.

The source sends the SRI at the rate set by the LBM configuration option **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)**. The Stores must receive this SRI.

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 instance(s).

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 instance. 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 Store instance(s) 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 instance(s). 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.
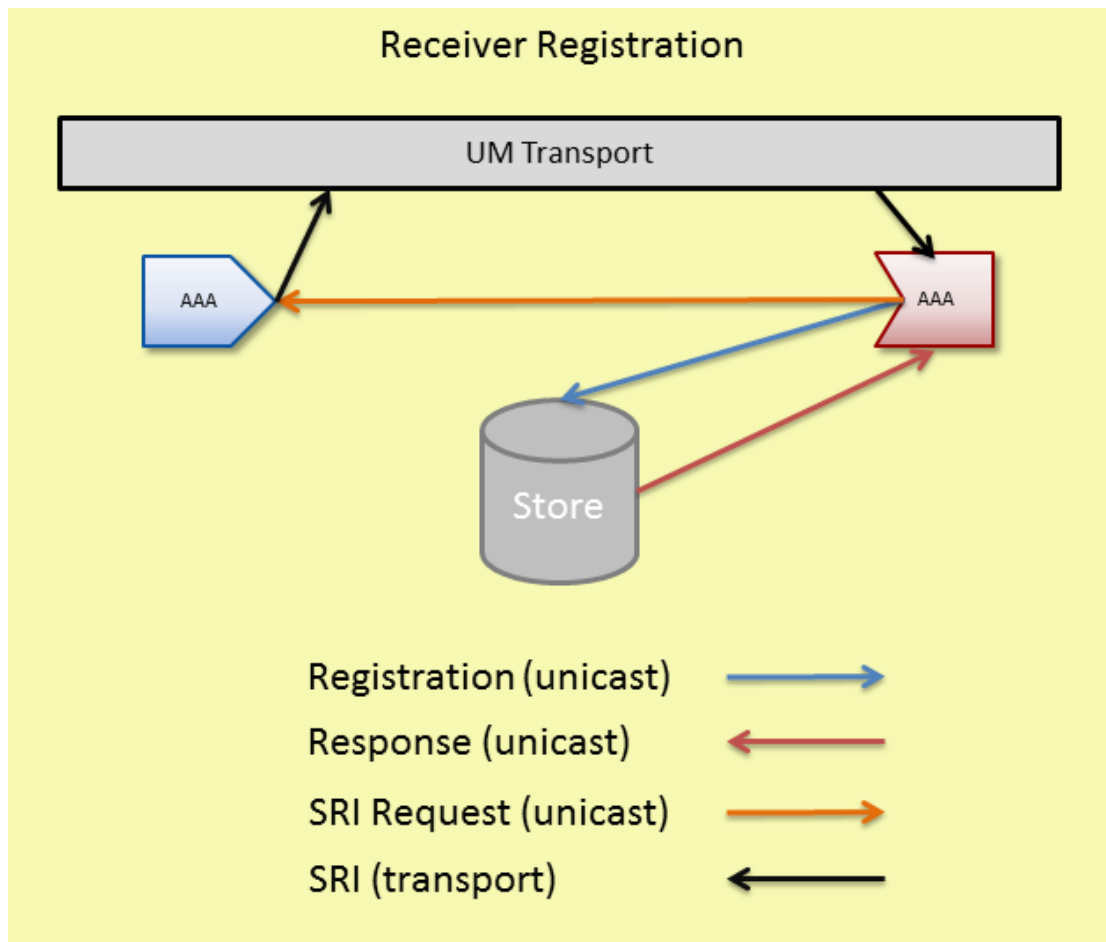
5.2.1　Receiver Registration Process

Any receivers who have resolved their topic and joined the transport session when the source sends out SRIs can register with the Store instance. 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 instance. 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.
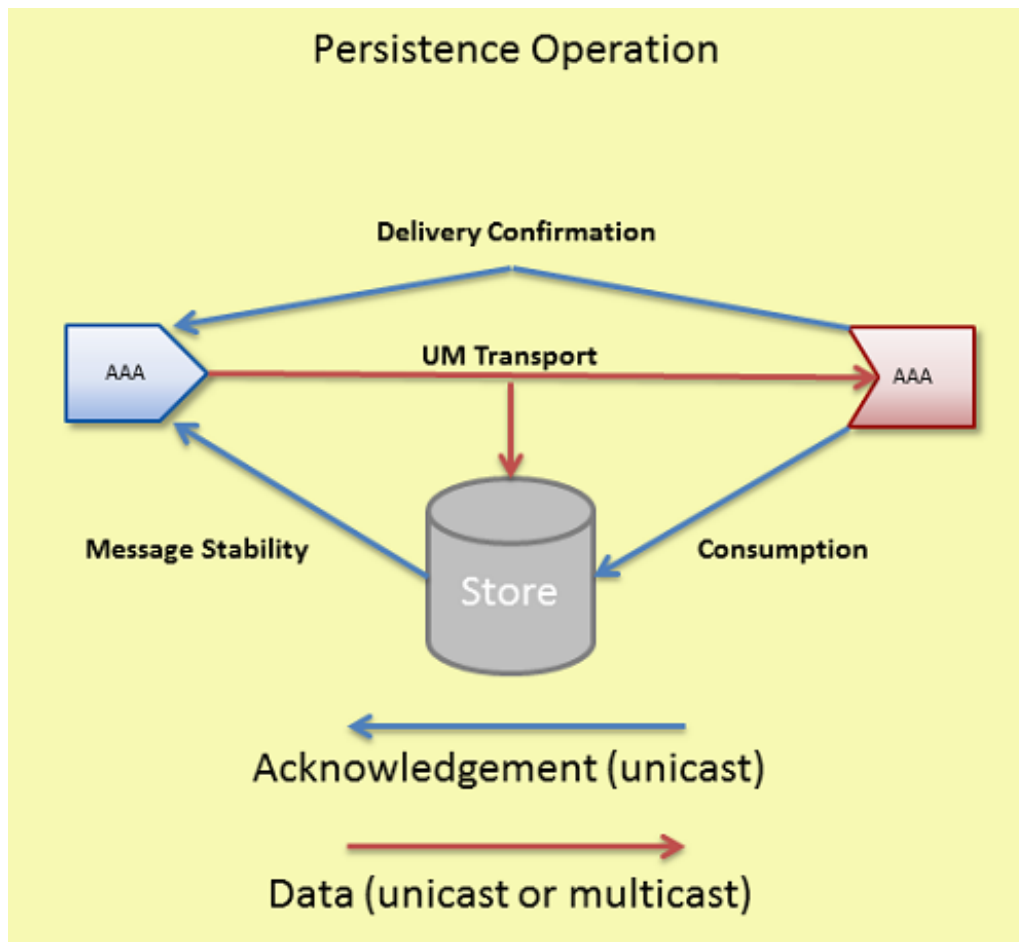
During normal persistence operation:

1. Sources transmit data to receivers and Stores at the same time over UM multicast or unicast transport protocols.

2. As the Store instance receives and persists messages, it 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 (LBM configuration options **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 instance(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 Store instances 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_EWOULDBLOCK.

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, ac-
cording 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 **DROs**.

The following diagram illustrates receiver recovery:
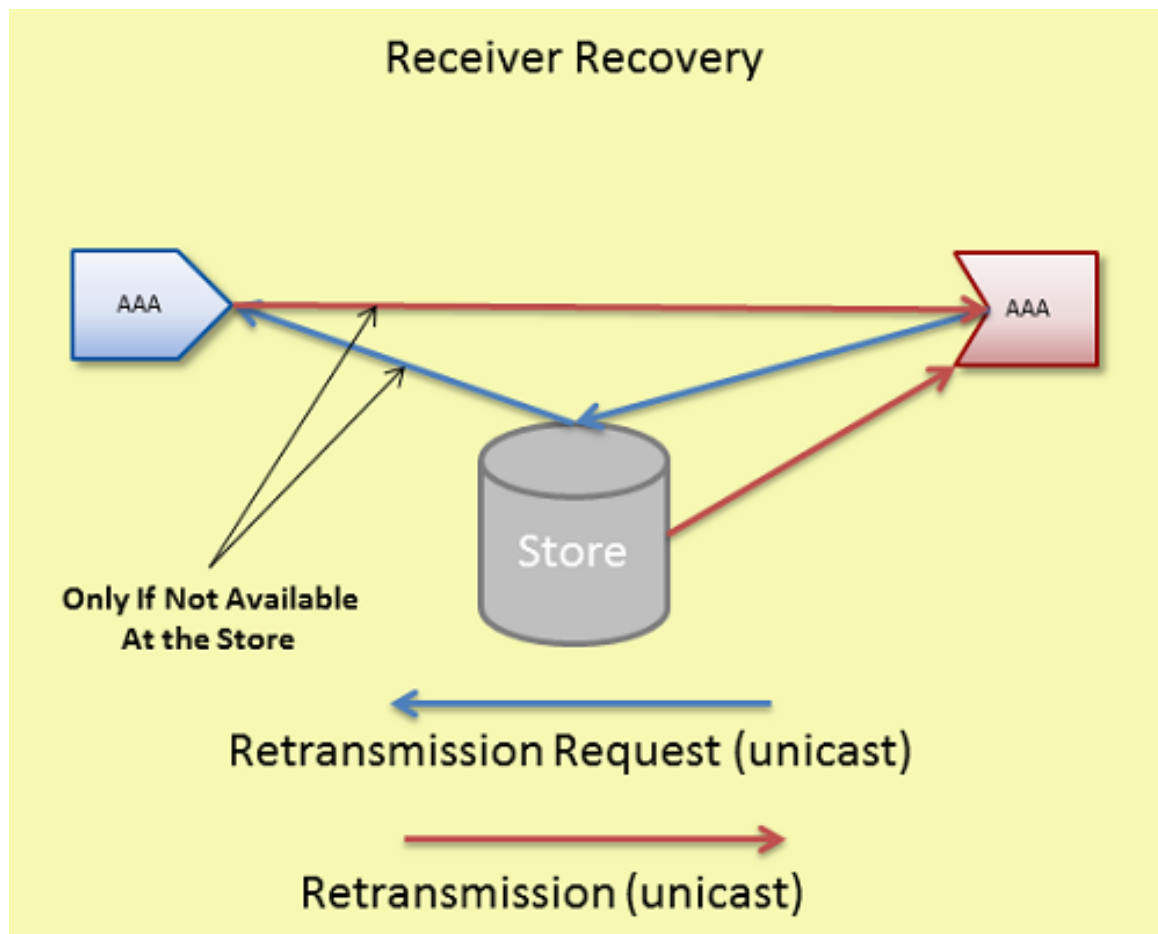
Receivers unicast retransmission requests. If the Store has the message, it unicasts the retransmission to the receiver. If it does not have the message and is configured to forward the request to the source, it unicasts the retransmission request to the source. If the source has the message, it unicasts the retransmission directly to the receiver. See also Message Loss Recovery.

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

5.2.5 Registration Limitations

An important use case for UM Persistence is the idea that an application registers, either with a RegID or a Session↩ID, and can then exit (gracefully or not) and subsequently it can re-register with the same RegID or SessionID and pick up where it left off.

This re-registration has some limitations regarding operational parameters changing between the registration and the re-registration. In general, an application re-registering a source or receiver should use the same operational parameters that it used when it originally registered.

In particular, except as noted below, the re-registering application should use the same values for any "ume_..." configuration options supplied.

There are some exceptions to this rule:

- It is permissible for an application bind to a different IP address and/or Port. This is important because a failure might render the original host unusable, so the application must be allowed to migrate to a different host.

- It is permissible for the application to use a different transport type (TCP, LBT-RM, LBT-RM, IPC, etc). This is important because a migration to a different host might impose different networking restrictions (e.g. no multicast).

- The values for **ume_store (source)** can change (IP/Port/TRD). This is important because a Store might fail and need to be migrated to a different host.

5.3 RPP: Receiver-Paced Persistence

The Receiver-paced Persistence (RPP) 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 LBM 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. If all receivers acknowledge within the write delay interval, the message is deleted from memory without having ever been written to disk. This gives an RPP Store comparable performance to a memory-only Store, while still giving a large disk-based repository if it is needed. (But notice that if slow or absent receivers cause the write delay to expire without the needed acknowledgements, the Store performance will return to the general performance of an SPP Store. You can tell if the Store has resorted to writing to disk by looking at the file size of the cache file. If it is greater than zero, it represents a high water mark of data written to disk since the file was last deleted.

Informatica recommends provisioning Stores based on SPP Store performance.

RPP introduces the capability of a source application to override set the settings for the following operational options on the Store:

- repository-size-threshold

- repository-size-limit

- repository-disk-file-size-limit

- repository-disk-write-delay

- repository-allow-ack-on-reception (The source doesn't actually override this option. This option enables the source to set the Store's ACK behavior.)

With SPP, those parameters are set only by the Store 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.
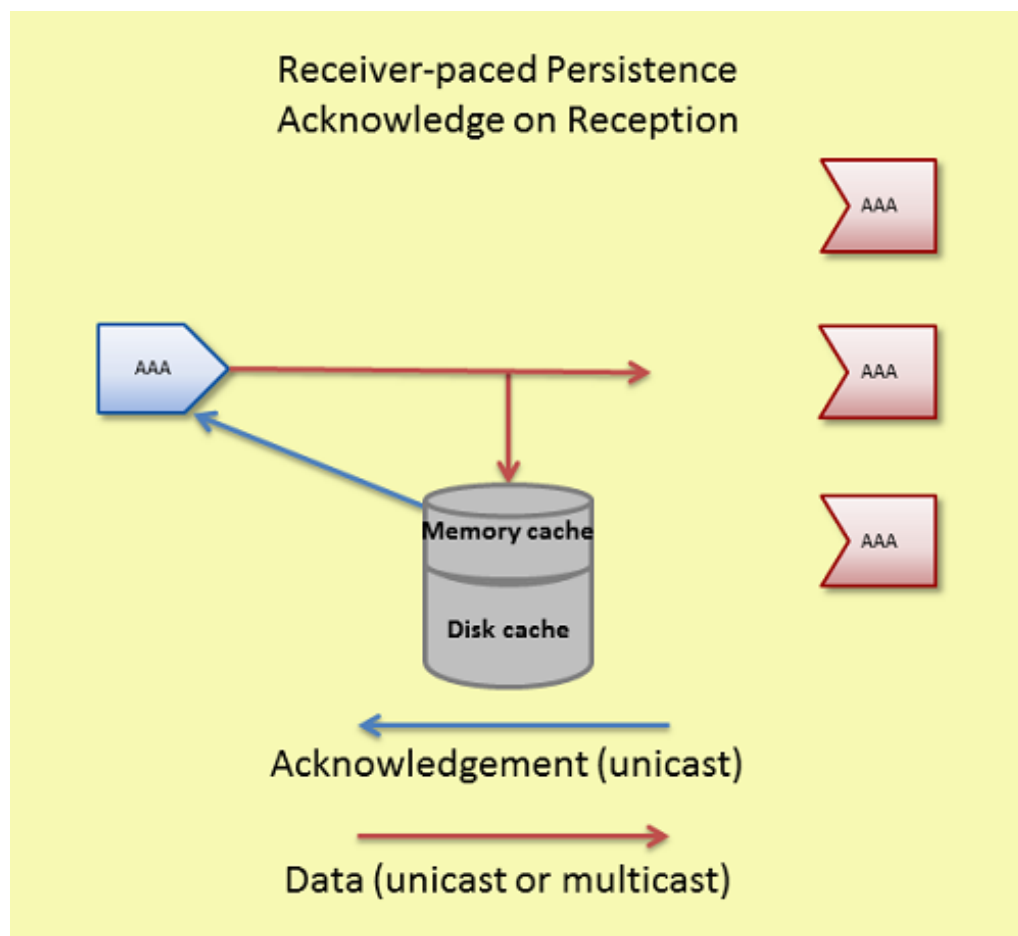
- Write Delay - The repository option repository-disk-write-delay allows the repository to hold messages in memory cache longer before persisting them to disk. This delay increases the probability that all RPP receivers acknowledge message consumption, eliminating the need to persist the message to disk.

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 available 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 LBM 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 Store 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"**), 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 LBM 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 an LBM configuration file. Although only two files appear below, this configuration represents two, single-Store quorum/consensus groups and one UM context. A second Store configuration file would be required for the Store "store1rpp" containing options and values identical to "store0rpp".

LBM Configuration File for RPP

The following example LBM 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 LBM 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
```

Store 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 LBM configuration files similar to that shown in the preceding section.

There are actually three Stores configured in QC. 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-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="2048"/>
                <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	
DRO	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	

UM Feature	Supported	Notes
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. See **LBM_SRC←_EVENT_UME_REGISTRATION_SUCCESS_EX**.
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. See **LBM_SRC_EVENT_UME_REGISTRATION_COMP←LETE_EX**.
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. See **LBM_SRC_EVENT_UME_REGISTRATION←_ERROR**.
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. See **LB←M_SRC_EVENT_UME_MESSAGE_STABLE_EX**.

Event Name	Type	Description
Store Message Not Stable	Source Event	Delivered once a message's **ume_message_stability_↩ lifetime (source)** has expired. The source no longer re-transmits the message to the Store. See **LBM_SRC_EV↩ ENT_UME_MESSAGE_NOT_STABLE**.
Delivery Confirmation	Source Event	Delivered once a message has been confirmed as deliv-ered and processed by a receiving application. Event con-tains the message sequence number as well as indications whether the message has met the unique confirmations re-quirement. Also contains the receiver's Registration ID or Session ID. See **LBM_SRC_EVENT_UME_DELIVERY_↩ CONFIRMATION_EX**.
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 applica-tion will not be allowed to send messages. See **LBM_SR↩ C_EVENT_UME_STORE_UNRESPONSIVE**.
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.) See **LBM_SRC_EVENT_↩ UME_MESSAGE_RECLAIMED_EX**.
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. See **ume_force↩ _reclaim_function (source)**.
Flight Size Notification	Source Event	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 mes-sage recently stabilized has reduced the number of in flight messages to less than the flight size limit (UNDER). See **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION**.
Source Deregistration Success	Source Event	Delivered once a source successfully deregisters from an in-dividual Store. The event contains either the RegID or Ses-sion ID, the sequence number of the last message stored for the source and Store information. See **LBM_SRC_EV↩ ENT_UME_DEREGISTRATION_SUCCESS_EX**.
Source Deregistration Complete	Source Event	Delivered once UM receives a successful deregistration event from all Stores. See **LBM_SRC_EVENT_UME_D↩ EREGISTRATION_COMPLETE_EX**.

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. See **LBM_MSG_U↩ME_REGISTRATION_SUCCESS_EX**.
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. See **LBM_MSG_UME_REGISTRATION↩_COMPLETE_EX**.
Store 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. See **LB↩M_MSG_UME_REGISTRATION_ERROR**.
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. See **LBM_MSG_UME_DEREGI↩STRATION_SUCCESS_EX**.
Receiver Deregistration Complete	Message	Delivered once UM receives a successful deregistration event from all Stores. See **LBM_MSG↩_UME_DEREGISTRATION_COMPLETE_EX**.
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. See **LBM_MSG_UME_REGISTRATIO↩N_CHANGE**.
Store Retransmission	Message	Retransmissions from recovery come in as normal messages with a flag indicating their status as a retransmission. See **LBM_MSG_FLAG_↩UME_RETRANSMIT** and **LBM_MSG_FLAG_↩OTR**.
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. See **ume_registration_extended_↩function (receiver)**.

Event Name	Type	Description
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. See **ume_recovery_sequence_↩ number_info_function (receiver)**.

5.5 Store Monitoring

See **Monitoring** for an overview of monitoring an Ultra Messaging network.

It is important to the health and stability of a UM network to monitor the operation of Stores. This monitoring should include real-time automated detection of problems that will produce a timely alert to operations staff.

Three types of data should be monitored:

- Log file.

- UM library statistics (context, source, receiver, wildcard receiver, event queue).

- Daemon statistics (similar data to the Store Web Monitor).

For UM library stats and daemon stats, the monitoring messages contain an "application ID". For UM applications, this is a user-specified name intended to identify the individual component/instance, and is supplied by the option **monitor_appid (context)**.

However, in the Store, the "monitor-appid" is typically set in the store's XML configuration file, so that it can be set on a store-basis. I.e. if the Store daemon is configured with multiple Store instances, each one can be given its own "monitor-appid".

For example, a Store configured with:

```
<stores>
  <store name="store_topic1" interface="10.29.4.0/24" port="12801">
    <ume-attributes>
      <option type="lbm-context" name="monitor_appid" value="store_appid_topic1"/>
       ...
```

The UM library stats will have the application ID "`store_appid_topic1`".

However, in the case of Store daemon stats, the "name" attribute of the UMP Element "<store>" is used as the application ID. So in the above example Store configuration, the daemon stats will have the application ID "`store↩ _topic1`".

5.5.1 Store Monitoring: Logs

Ideally, log file monitoring would support the following:

- Archive all log messages for all Stores for at least a week, preferably a month.

- Provide rapid access to operations staff to view the latest log messages from a Store.

- Periodic scans of the log file to detect errors and raise alerts to operations staff.

Regarding log file scanning, messages in the Store's log file contain a severity indicator in square brackets. For example:

```
Sun Oct 30 08:32:22 2022 [WARNING]: Store-5688-5445: WARNING: store "store_topic1"
    cache directory appears to be on an NFS mount. This is not recommended.
```

Informatica recommends alerting operations staff for messages of severity [WARNING], [ERROR], [CRITICAL], [ALERT], and [EMERGENCY].

It would also be useful to have a set of exceptions for specific messages you wish to ignore. For example, if you have a Store that must locate its state and cache files on an NFS mount, you might want to have your scanner exclude message Store-5688-5445.

There are many third party real-time log file analysis tools available. A discussion of possible tools is beyond the scope of UM documentation.

5.5.2 Store Monitoring: UM Library Stats

The Store communicates with persistent applications using Ultra Messaging protocols, and therefore makes use of the UM library. It is just as important to monitor the UM library statistics for the Store as it is for applications. Automatic monitoring is enabled using the same configuration options as for applications.

There are two data formats for UM library stats:

- Protobufs - recommended.

- CSV - deprecated. Informatica recommends migrating to protobufs.

For example, here is an excerpt from a sample Store configuration file that shows how an XML-based UM library configuration file is specified:

```
<ume-store version="1.3">
  <daemon>
    ...
    <xml-config>um.xml</xml-config>
    ...
```

Here is an excerpt from a sample "um.xml":

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-configuration version="1.0">
  <templates>
    ...
    <template name="automonitor">
      <options type="context">
        <option name="monitor_format" default-value="pb"/>
        <option name="monitor_interval" default-value="600"/>
        <option name="monitor_transport" default-value="lbm"/>
      </options>
    </template>
    ...
    <template name="mon_ctx">
      <options type="context">
        <option name="resolver_unicast_daemon" default-value="10.29.3.101:12801"/>
        <option name="default_interface" default-value="10.29.3.0/24"/>
        <option name="mim_incoming_address" default-value="0.0.0.0"/>
        ...
      </options>
      <options type="source">
        <option name="transport" default-value="tcp"/>
      </options>
    </template>
    ...
  </templates>
```

```
<applications>
  ...
  <application name="umestored">
    <contexts>
      <context name="store_topic1" template="mytemplate,automonitor">
        <sources/>
      </context>
      <context name="29west_statistics_context" template="mon_ctx">
        <sources/>
      </context>
    </contexts>
  </application>
  ...
```

Notes:

1. The **monitor_format (context)** value "pb" selects the protobuf format and is available for the Store in UM version 6.14 and beyond. Selecting this format implicitly enables the inclusion of the Store's daemon stats (see below).

2. For a list of possible protobuf messages for the Store, see the "ump_mon.proto" file at **Example ump_mon.↩ proto**.

3. The Store context named "store_topic1" is configured with the `"automonitor"` template, which sets the automatic monitoring options. The **monitor_interval (context)** option enables automatic monitoring and defines the statistics sampling period. In the above example, 600 seconds (10 minutes) is chosen somewhat arbitrarily. Shorter times produce more data, but not much additional benefit. However, UM networks with many thousands of applications may need a longer interval (perhaps 30 or 60 minutes) to maintain a reasonable load on the network and monitoring data storage.

4. When automatic monitoring is enabled, it creates a context named "29west_statistics_context". It is configured with the "mon_ctx" template, which sets options for the monitoring data TRD. (Alternatively, you can configure the monitoring context using **monitor_transport_opts (context)**.) When possible, Informatica recommends directing monitoring data to an administrative network, separate from the application data network. This prevents monitoring data from interfering with application data latency or throughput. In this example, the monitoring context is configured to use an interface matching `10.29.3.0/24`.

5. In this example, the monitoring data **TRD** uses **Unicast UDP Topic Resolution**. The **lbmrd** daemon is running on host 10.29.3.101, port 12001.

6. The monitoring data is sent out via UM using the TCP transport.

7. These settings were chosen to conform to the recommendations in **Automatic Monitoring**.

For a full demonstration of monitoring, see: `https://github.com/UltraMessaging/mcs_demo`

5.5.3 Store Monitoring: Daemon Stats

The daemon statistics for the Store represent a superset of the information presented on the Store Web Monitor.

There are two data formats for the Store to send its daemon stats:

- **Protobufs** - recommended. The protobufs format is accepted by the **Monitoring Collector Service (MCS)** and the "lbmmon" example applications: **Example lbmmon.c** and **Example lbmmon.java**.

- **Binary** - deprecated. Informatica recommends migrating to protobufs. For information on the deprecated binary formatted daemon stats, see Store Binary Daemon Statistics.

The recommended way to enable Store daemon stats is by enabling UM library stats using using **monitor_format (context)** with format "pb". For example, here's an excerpt from a UM library XML configuration file from `https↩ ://github.com/UltraMessaging/mcs_demo` file `um.xml`:

```
<template name="automonitor">
  <!-- Additional application configuration to enable automatic monitoring. -->
  <options type="context">
    <option name="monitor_format" default-value="pb"/>
    <option name="monitor_interval" default-value="600"/>
    <option name="monitor_transport" default-value="lbm"/>
  </options>
</template>
...
<application name="store1">
  <contexts>
    <context name="store_topic1" template="mytemplate,automonitor,res_trd1">
      <sources/>
    </context>
    <!-- Context created by automatic monitoring. -->
    <context name="29west_statistics_context" template="mon_ctx">
      <sources/>
    </context>
  </contexts>
</application>
```

The protobufs format is accepted by the **Monitoring Collector Service (MCS)** and the "lbmmon" example applications: **Example lbmmon.c** and **Example lbmmon.java**.

For a list of possible protobuf messages for the Store, see the "ump_mon.proto" file at **Example ump_mon.proto**.

For a full demonstration of monitoring, including Store daemon stats, see: [https://github.com/Ultra←](https://github.com/Ultra) [Messaging/mcs_demo](https://github.com/UltraMessaging/mcs_demo)

See also [Store Monitoring: UM Library Stats](#).

Chapter 6

Store Repository Profiling (SRP)

To aid the users in operating and maintaining their Persistent Store deployment, Informatica provides an API that will read Store cache and state files and return useful information about them, including message content. This is the Store Repository Profiling (SRP) API. (The API is not in the normal "lbm" library; see Using the SRP API.)

Also supplied is the "umesnaprepo" example program, which uses the SRP API to read the information and print to standard out. It can be used as-is (see umesnaprepo Man Page), or its source code (**Example umesnaprepo.c**) can be used as a guide for users to develop their own management tools.

This API and example program are supported on the same platforms that support the Persistent Store: 64-bit Linux and 64-bit Windows. The API is C-only (no Java or .NET). Also be aware that the API is read-only. The API does not provide a way to modify the cache or state files.

IMPORTANT: due to differences in certain Windows and Linux data sizes, a given set of state and cache files needs to be read on the same platform where it was generated. For example it is *NOT* valid to copy a Linux Store's files to a Windows machine and use the windows-based SRP API or command-line tool to read them.

Note

> The Store instance **should NOT be running** while the API or example program is used to read the Store files. There currently is no tool that performs the same function on an actively running Store.

6.1   Using the SRP API

Use the **Example umesnaprepo.c** source code as your guide.

The umeprofile.h header file contains the needed definitions.

The API code is *not* contained within the normal "lbm" library. On Linux, it is in "libumestorelib.a", a static library. On Windows, it is in "umestore.dll", a dynamic library.

The main API functions are:

- **lbm_srp_create()**
- **lbm_srp_delete()**
- **lbm_srp_get_repo_state()**
- **lbm_srp_free_repo_state()**
- **lbm_srp_get_repo_message()**

6.2 umesnaprepo Man Page

The "umesnaprepo" command is an example program that uses the Store Repository Profiling (SRP) API to read Store state and cache files and print useful information, including message content. It can be used as-is, or its source code (**Example umesnaprepo.c**) can be used as a guide for users to develop their own management tools.

Note

> The Store instance **should NOT be running** while the API or example program is used to read the Store files. There currently is no tool that performs the same function on an actively running Store.

```
Usage: umesnaprepo -s state_dir [options]
Available options:
  -c, --cache-dir=PATH     cache file search PATH
  -h, --help               display this help and exit
  -n, --no-checksum        disable cache checksum checking
  -p, --parse              enable LBM header parsing
  -s, --state-dir=PATH     state file search PATH [required]
  -t, --truncate=NUM       limit cache message displays to NUM bytes
  -T, --terse              summarize cache and skip cache message displays
```

For example:

```
umesnaprepo -s /UM/store5/state -c /UM/store5/cache
```

This examines the state and cache files and prints information for every Store instance represented there. Here's some example output:

```
Examining repository at index: 0
state_filename: /UM/store5/state/2545027182-state
cache_filename: /UM/store5/cache/2545027182-cache
...
Repository cache:
    number of messages: 26
...
Receiver 0
    regid: 2545027183
    sqn: 25
...
Message sqn [0]:
   tsp: 1584022054.898262
   disk_len: 37
   disk_offset: 0
   flags: 0x01
Message body:
00 00 00 25 a4 ab 82 e6 00 00 00 00 6d 65 73 73        ...%........mess
61 67 65 20 30 00 00 00 00 00 00 00 00 00 00 00        age 0...........
00 00 00 00 00                                         .....
...
```

In this example output, notice that the actual beginning of the message (the first byte is the "m" of "message 0") is at offset 12 from the beginning of the buffer. The 12 bytes in front of the message is the "LBMC" header. The example application lets you supply the "-T" option, which parses the LBMC header and starts printing the message at the actual start of data.

Chapter 7

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

7.1 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

7.2 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 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 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.

7.3 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 LBM 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.

7.4 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.

7.5 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`

7.5.1 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)**

7.6 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 8

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 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.

```
Sun Oct 29 14:15:01 yyyy [INFO]: Store-5688-5273: Latency Busters Persistent Store
    version 6.17
Sun Oct 29 14:15:01 yyyy [INFO]: Store-5688-5274: UMP 6.17 [UMP-6.17] [64-bit]
    Build: mmm dd yyyy, hh:mm:ss ( DEBUG license LBT-RM LBT-RU LBT-IPC LBT-SMX )
    WC[PCRE 7.4 2007-09-21, regex, appcb] HRT[gettimeofday()]
```

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 Store. See Store Web Monitor.

8.1 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-Reg←
ID' 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, UM←
E-example-rcv-RegID.

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

8.2 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.

8.3 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.

8.4   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 9

Registration Identifiers

As mentioned in Registration Identifier Concept and Adding Fault Recovery with Registration IDs, Stores use Reg←
IDs 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.

9.1 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)** LBM configuration option. See also ume-example-src-2.c

C API

```
lbm_src_topic_attr_t * sattr;

if (lbm_src_topic_attr_create_from_xml(&sattr, "MyCtx", src_topic_name) == LBM_FAILURE) {
  fprintf(stderr, "lbm_src_topic_attr_create_from_xml: %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);
}
```

9.2 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. */
```

9.3   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 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 LBM 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.

Chapter 10

Designing Persistent Sources

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

10.1 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_from_xml(&attr, "MyCtx", src_topic_name);
    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.

10.2 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;
}
```

10.3 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 turns off **ume_message_stability_notification (source)**, **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, re-
  maining 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.

10.4   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 **Example 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;
```

10.5 Source Retention Policy Options

Sources use a set of configuration options to release messages that, in effect, specify the source's retention 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)**

10.6 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_from_xml(&sattr, "MyCtx", src_topic_name) == LBM_FAILURE) {
    fprintf(stderr, "lbm_src_topic_attr_create_from_xml: %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);
}
```

10.7 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

```c
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

```java
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;
}
```

10.8 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 option is enabled */
{
    lbm_src_event_ume_ack_ex_info_t *info = (lbm_src_event_ume_ack_ex_info_t *)ed;
```

```c
    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 QC 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 option 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

```java
case LBM.SRC_EVENT_UME_MESSAGE_STABLE_EX:
    // requires that source ume_message_stability_notification option 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 QC 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 option 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 option 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 QC 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 option 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;
```

10.9   Mapping Your Message Numbers to Sequence Numbers

Some application developers want their publishing applications to know the UM-assigned topic-level sequence numbers assigned to outgoing messages.

The C API functions **lbm_src_send_ex()** and **lbm_src_sendv_ex()** allow you to expand the number of source events that will be delivered to your sending application, including a source event that informs your sender of the topic-level sequence numbers assigned to each message you send. In the case of a large message that requires **fragmentation**, the callback will tell you the starting and ending topic-level sequence numbers assigned to it.

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;
struct my_sqn_nums_s{
  unsigned int first_sqn;
  unsigned int last_sqn;
};
typedef struct my_sqn_nums_s my_sqn_nums_t;
my_sqn_nums_t msg_sqn_nums;

...

/* 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 = &msg_sqn_nums;
if (lbm_src_send_ex(src, message, msglen, 0, &exinfo) == LBM_FAILURE) {
    ... /* handle error. */
}
/* Before lbm_src_send_ex returns, the source event will be delivered on the callers thread,
 * setting up msg_sqn_nums. */
printf("first_sqn=%u, last_sqn=%u\n", msg_sqn_nums.first_sqn, msg_sqn_nums.last_sqn);
```

C API - Source 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;
            my_sqn_nums_t *sqn_nums = (my_sqn_nums_t *)info->msg_clientd;

            sqn_nums->first_sqn = info->first_sequence_number;
            sqn_nums->last_sqn = info->last_sequence_number;
        }
        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
try {
    src.send(message, msglen, 0, exinfo);
}
catch(UMENoRegException ex) {
    // Handle error
}
// Access start/end sqns via instance variables.
```

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();
            // Set instance variables to info.firstSequenceNumber() and info.lastSequenceNumber().
            break;
        }
        ...
    }
    return 0;
}
```

.NET API - Enable Message Information

.NET code is the same as Java (above).

10.10   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 **DRO**.

- 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.

Chapter 11

Designing Persistent Receivers

Receivers are predominantly interested in RegID management and recovery management.

11.1   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.

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;
```

```c
    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);
            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

```java
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() + "] ");
        }
        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() + "] ");
        }
        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;
}
```

11.2   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)**.

11.3 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.

11.4   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;
    }
}
```

11.5   Persistence Message Consumption

When a persistent subscriber application has finished processing a received message, it must signal consumption of the message to UM. This is how UM keeps track of where message recovery should begin if recovery is needed (e.g. if the receiver restarts).

There are three basic methods to signaling message consumption that the application must choose between:

- The subscriber's receiver callback returns to UM, allowing UM to delete the message. The message deletion signals consumption. This is generally the most simple case.

- The subscriber retains the message past the point of the receiver callback returning. Then some other application function (typically running in a separate thread) finishes the processing and deletes the message. The message deletion signals consumption.

- The subscriber calls a separate "explicit ACK" API which signals message consumption. In this case, the act of deleting the message does not signal consumption.

When the application uses one of those methods to signal consumption, it is informing the local instance of UM running within the application. As a separate step, UM must send a consumption acknowledgement (ACK) to the Persistent Store(s). However, the two steps are not necessarily directly linked.

There are two configurable acknowledgement methods that UM can follow when the application signals consumption:

- UM batches ACKs based on a configurable time period. I.e. UM will delay sending ACKs to the Store so that multiple messages can be acknowledged at once, when its timer expires.

- UM immediately sends an ACK to the Store.

Batching of ACKs greatly improves the throughput of a persistent receiver, and can also improve latency. It is the default configuration.

However, batching has the disadvantage that if message recovery is needed (e.g. if the subscriber restarts), it increases the chances that already processed messages will be re-sent during recovery ("duplicate messages").

Any messages that the application signaled consumed, but which UM has not yet acknowledged to the Store, can be re-sent during recovery.

Note that even with immediate ACK, there is still the possibility of one or more already-consumed messages being recovered; applications need to be able to handle this. See Duplicate Message Delivery for more information.

Use Cases

There are four common use cases which combine an application consumption method with a UM acknowledgement method:

- Delete on Return, Batch ACKs, most common.

- Retain on Return, Batch ACKs, also common.

- Explicit Acknowledgments.

- ACK Immediately on Delete.

11.5.1 Delete on Return, Batch ACKs

The application receiver callback completes all processing of a message before returning. It allows UM to delete the message on return, which signals consumption.

To maximize efficiency, the application allows UM to batch ACKs to the Store.

Note that in this use case, messages are always signaled consumed in the same order in which they are received.

For this use case, use the configuration:

- **ume_use_ack_batching (receiver)** - set to **1** (the default).

- **ume_explicit_ack_only (receiver)** - set to **0** (the default).

The subscriber code is written to signal message consumption on return from the receiver callback. This is handled differently between the C API vs. the Java and .NET APIs.

C API

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

Note: the receiver callback must *not* call **lbm_msg_delete()**.

Java and .NET

With Java and .NET, the application receiver callback must explicitly call **com::latencybusters::lbm::LBM←Message::dispose()** prior to returning.

(Note that this is not strictly true for .NET. A .NET program can skip calling "dispose()" and allow the message to become garbage. This will introduce significant latency outliers when GC runs, and also makes acknowledgements to the Store non-deterministic. Finally, in the future, performance improvements for .NET will probably require the use of "dispose()". Informatica recommends that .NET programs call "dispose()" for every message.)

11.5.2 Retain on Return, Batch ACKs

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 handed off to a worker thread for longer-term processing.

Or the acknowledgement must be delayed until some asynchronous event happens, like a handshake with another application.

To maximize efficiency, applications allow UM to batch ACKs to the Store.

Note that some applications are written to process these handed-off messages in parallel, and messages might be signaled consumed in a different order than they arrived. In this use case, out-of-order consumption is supported. See ACK Ordering for more information.

For this use case, use the configuration:

- **ume_use_ack_batching (receiver)** - set to **1** (the default).

- **ume_explicit_ack_only (receiver)** - set to **0** (the default).

Note that this is the same configuration as Delete on Return, Batch ACKs.

The subscriber code should be written to hand off the message to another part of the application and suppress deleting the message on return from the receiver callback. This is handled differently between the C API vs. the Java and .NET APIs.

C API

In C, the application's receiver callback function must call the **lbm_msg_retain()** API for the received message prior to handing off the message for processing. This suppresses the automatic deletion of the received message when the receiver callback returns, and allows the message buffer to be passed 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, it signals consumption by calling **lbm_msg_delete()**.

Java and .NET

In Java and .NET, the application should call the **com::latencybusters::lbm::LBMMessage::promote()** API prior to handing the message for a separate processing. This allows the message object to be passed to some other part of the application for processing and disposal at a later time.

When the application subsequently completes all processing of the message, it signals consumption by calling **com::latencybusters::lbm::LBMMessage::dispose()**.

11.5.3 Explicit Acknowledgments

The **ACK batching feature** did not exist in UM in versions prior to 5.0. In those early versions, the application had to use the **Explicit ACK feature** to benefit from the increased throughput allowed by batching of ACKs. Informatica will continue to support Explicit ACKs, even though we generally recommend the ACK batching use cases Delete on Return, Batch ACKs or Retain on Return, Batch ACKs.

With explicit ACKs, message consumption is separated from message deletion. The application uses a separate API to trigger an acknowledgement to the Store. Batching is achieved under application control by not ACKing every message. With explicit ACKs, the action of acknowledging a message to the Store implicitly acknowledges all previous unacknowledged messages.

Warning

Because explicit ACKs implicitly acknowledge all previous unacknowledged messages, it is not valid to acknowledge messages out of order. Otherwise the application risks not recovering needed messages. See ACK Ordering.

For this use case, use the configuration:

- **ume_use_ack_batching (receiver)** - set to **0**.

- **ume_explicit_ack_only (receiver)** - set to **1**.

The subscriber should be written the same as Delete on Return, Batch ACKs (if processing is completed in the receiver callback) or Retain on Return, Batch ACKs (if the message is retained after the receiver callback returns). The same coding rules apply regarding message retention and disposal.

However, as an additional step, the application must call the explicit ACK API:

- C: **lbm_msg_ume_send_explicit_ack()**

- Java/.NET: **com::latencybusters::lbm::LBMMessage.sendExplicitAck()**

This is typically done inside of a timer callback, which provides batching behavior.

11.5.4   ACK Immediately on Delete

With this use case, the application directs UM to send an acknowledgement to the Store for each and every processed message immediately after the message is deleted.

The advantage of this use case is that it minimizes the chances that already processed messages will be resent during recovery (i.e. "duplicate messages"). However, there is still the possibility of one or more duplicate messages during recovery, so applications need to be able to handle this. See Duplicate Message Delivery for more information.

A disadvantage of this use case is that the maximum sustainable throughput (message rate) is limited by the per-message overhead of sending acknowledgements.

Another disadvantage is that messages must be deleted in the same order as they were received, even if the messages are handed off to another thread for processing. See ACK Ordering.

Informatica's general recommendation is that since the application needs to handle duplicate messages even with immediate acknowledgement, the user should implement a use case that includes ACK batching.

For this use case, use the configuration:

- **ume_use_ack_batching (receiver)** - set to **0**.

- **ume_explicit_ack_only (receiver)** - set to **0** (the default).

The subscriber should be written the same as Delete on Return, Batch ACKs (if processing is completed in the receiver callback) or Retain on Return, Batch ACKs (if the message is retained after the receiver callback returns). The same coding rules apply regarding message retention and disposal.

11.6   ACK Ordering

When UM is allowed to batch ACKs to the Store using **ume_use_ack_batching (receiver)**, the UM client library supports "out of order" signaling that the application is done processing messages. For example, the application might retain received messages 1, 2, and 3 for asynchronous processing, and then delete them in the order 2, 1, 3.

However, be aware that 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.

The UM client library handles this by withholding acknowledgement to the Store all messages for which an earlier message is not signaled complete. For example, if the application retains messages 1, 2, and 3, and subsequently

deletes message 2, UM will not send an ACK to the Store for message 2. If the application then deletes message 1, UM is now able to send an ACK for message 2, which will implicitly acknowledge message 1 as well.

However, this ability to process messages out of order is dependent on using ACK batching (i.e. **ume_use_ack↩ _batching (receiver)** is **1**). If ACK batching is disabled, then the application is responsible for ordering acknowledgements to the Store. This is true for both the Explicit Acknowledgments and ACK Immediately on Delete use cases.

For example, suppose that the ACK Immediately on Delete use case is being used. If the application processes and deletes messages in the order 2, 1, 3, the ACK for message 2 will implicitly ACK message 1 to the Store. If the application crashes after ACKing 2 and restarts, recovery will start with message 3. Thus, message 1 is never fully processed.

If ACK batching is disabled, the messages must be deleted in order: 1, 2, 3.

11.7 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. */
}
```

Chapter 12

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.

This document is oriented mostly to programmers. See also the Operations Guide chapters **Running Persistent Stores (umestored)**, **Persistent Store Crashed**, **Persistent Sending Problems**, and **UM Persistent Store Log Messages**.

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 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.

12.1   Limit Initial Restore with Restore-Last

The "restore-last" feature limits the initial message restore for a restarting Store.

When a disk-based store is restarted, during its initialization it will open the state and cache files and restore the data. This makes previously saved message data available for recovering subscribers. Note that this message restoration takes some time. For small files of a few megabytes, it might take a few seconds. But for large files of many gigabytes, it could take minutes.

Starting with UM version 6.15, you can configure a restarting Store to only restore a subset of the saved messages. This can greatly speed up the process of initialization.

For example, you might direct the Store to only restore 8 hours' worth of message data when it is restarted. Doing this means that older messages are not available from this Store for recovering subscribers. But if only one Store of a Q/C group is restarted, one or more of the other Stores will continue to have the older messages available.

The UMP Element "<restore-last>" is used to enable this feature.

When the restore-last feature is enabled, the Store will write an additional, per-source "cache index" file. It is written to the configured cache directory and is named the same as the source's normal message cache file with ".idx" appended. For example, your cache directory might contain:

```
3085235048-cache
3085235048-cache.idx
```

When the Store is restarted, the cache index file is used to quickly determine which messages are within the restore-last range. As a result of this algorithm, if a deployed Store is re-configured to enable the restore-last feature, the first time it is restarted will not be sped up. I.e. since the index file will not yet exist, the first startup after enabling the feature will restore the entire existing cache file.

See UMP Element "<restore-last>" for implementation details.

12.2   Store Log File

The Store Process 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 UMP Element "<log>". 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.

12.3   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 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 UMP Element "<log>".

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.

12.4   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. The use of multiple QC groups is a special case and should be discussed with Informatica support before using.

12.5   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.

Chapter 13

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.

13.1 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.

13.2 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.

Chapter 14

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.

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).

See `umestored_main.h File Reference` for API details.

The API code is *not* contained within the normal "lbm" library. On Linux, it is in "libumestorelib.a", a static library. On Windows, it is in "umestore.dll", a dynamic library.

For an example of how to use the **umestored_main()** API, see the example program **Example 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 15

Store Thread Affinity

A significant performance improvement of the Store can be obtained by "pinning" threads to CPU cores. Normally, the operating system will migrate a process's threads to different CPU cores, depending on what else is going on in the host. This can degrade the process's performance in a number of ways, mostly related to memory access (cache, NUMA zones). By setting the CPU affinity for the performance-sensitive threads, you avoid this degradation.

For high-throughput applications, you will gain significant performance improvement by constraining the operating system to run the Store's threads on specific CPU cores. All of a Store's threads should run on cores in the same physical CPU chip.

For maximum benefit, you should "isolate" the cores running the message reception threads. This prevents the operating system from scheduling other processes/threads on those cores.

Setting Affinity

When the Store Process is executed, the user can optionally use the "-a" option to set CPU affinity to the various threads. See Umestored Man Page.

Note that for the Windows Service, you don't supply the option when the service is run. Instead you save the thread affinity into the Windows registry for subsequent use by the Store Windows Service. See Umestoreds Man Page and **Configure the Windows Service**.

The "-a" option takes a comma-separate list of CPU (core) numbers. For example, "-a 1,3,1,..." refers to CPU 1, CPU 3, CPU 1 again, etc.

The sequence of numbers are assigned to threads as follows:

The first number is the "process" CPU number, which is used for all miscellaneous threads that aren't otherwise assigned.

The next 4 numbers are assigned to a Store's operational threads in the following sequence:

1. Message reception thread.

2. Proxy source thread.

3. Receiver recovery thread.

4. Auxiliary thread.

If the Store Process has multiple Stores configured, additional groups of 4 numbers should be supplied.

Of these threads, the most critical is the message reception thread. For best performance, each Store's message reception thread should be given exclusive access to its own CPU core.

The receiver recovery thread is also important, since it can affect the speed at which receivers can recover missed messages. However, since CPU cores are scarce resources on hosts, it may not be practical to give each receiver recovery thread its own core.

The proxy source and auxiliary threads are not critical to general Store throughput, and are therefore generally assigned to the "process" core as miscellaneous.

Affinity Example

For example, suppose you have a Store Process configured for two Stores. Further, let's say that on your host, even-numbered CPUs belong to one physical CPU chip, and odd-numbered CPUs belong to a different physical CPU chip. The following would optimize both message reception and message recovery, at the expense of consuming 5 cores:

```
umestored -a 3,5,3,7,3,9,3,11,3 ...
```

This assigns:

  • the process's miscellaneous threads to CPU 3,

  • the first Store's message reception thread to CPU 5,

  • the first Store's proxy source thread to CPU 3,

  • the first Store's receiver recovery thread to CPU 7,

  • the first Store's auxiliary thread to CPU 3,

  • the second Store's message reception thread to CPU 9,

  • the second Store's proxy source thread to CPU 3,

  • the second Store's receiver recovery thread to CPU 11,

  • the second Store's auxiliary thread to CPU 3.

If assigning this many cores to the Store Process is not practical, the following conserves cores at the expense of potentially degrading message recovery speed:

```
umestored -a 3,5,3,3,3,7,3,3,3 ...
```

This assigns a CPU core to each of the two message reception threads (5 and 7), and groups all other threads onto the miscellaneous CPU core (3).

Chapter 16

Persistence Fault Tolerance

16.1 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.

16.2   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 create a UM source object to take the place of the exited source. This proxy source behaves much like a real source in that it provides all of the necessary information to subscribers so that they can discover and register with the Stores. This allows late joining receiver to recover messages they missed.

After the the real source returns, the Store automatically deletes its proxy source, allowing the real source to resume normal operation.

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.

16.2.1   How Proxy Sources Operate

The following sequence illustrates the life of a proxy source:

1. A source configured for Proxy Source sends to receivers and a group of Quorum/Consensus Stores.

2. The source fails.

3. The source's **ume_activity_timeout (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.

Note that the implementation of the proxy source involves the Store creating a normal UM source object. As such, the user is responsible for providing the Store with a UM library configuration with appropriate source-scoped options. For most source-scoped configuration options, there is no requirement for the proxy source's settings to match the original source's settings. However, there are a few that should be configured the same:

- **ume_retention_intergroup_stability_behavior (source)** (if configured by the original source).

- **ume_retention_intragroup_stability_behavior (source)** (if configured by the original source).

- **source-related topic resolution options** (e.g. **resolver_advertisement_minimum_sustain_duration (source)**).

Some UM customers have found reasons to intentionally configure their proxy source differently from the original source. For example, to conserve network resources, some customers choose to configure a different **transport** and change **topic-to-transport session mappings**. Feel free to contact UM Support for guidance in configuring your proxy sources.

If the Store running the proxy source fails, the other Stores in the Quorum/Consensus group detect a source failure again and can elect a new Store to initiate a proxy source, subject to the Store Option "proxy-source-repo-quorum-required".

16.2.2 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.

Activity Timeout

The Store uses the activity timer to decide if a new registration is allowed with the same registration ID. The Store does not allow two applications to be registered at the same time with the same registration ID. However, if an application exits abnormally, we obviously want to restart the application and have it register with the same registration ID. How does the Store prevent simultaneous registration while allowing sequential registrations? I.e. how does the Store decide that an existing registrant has exited? The activity timer.

After registration, the Store expects to hear some kind of activity (message, control, or keepalive) before the activity timer expires. If not, then the Store assumes the source or receiver has been deleted, perhaps by the program cleaning up, or perhaps by crashing. That "releases" the registration ID for use by another application instance.

Setting the activity timeout is somewhat of a balancing act. If you set it too long, then you need to wait a long time before you can restart a crashed application instance. If you set it too short, it risks the Store timing out the application too soon, leaving it vulnerable to having its registration ID "stolen" by another application instance.

Some users maintain tight control over their applications, and choose to set the activity timeout to zero. This results in "weak RegIDs", meaning that the Store does not enforce serialized access to the registration IDs. Other users choose a non-zero activity timeout, and rely on the Store to prevent simultaneous use of a registration ID. This results in "strong RegIDs", meaning that the Store enforces serialized access to the registration IDs.

The activity timeouts default to 30 seconds, and can be configured by the application using: **ume_activity_↩ timeout (source)** and **ume_activity_timeout (receiver)**. They can also be configured by the Store using: Topic Option "source-activity-timeout" and Topic Option "receiver-activity-timeout". (If both the application and the Store configures the same timer, the result varies and is described in the above linked documentation.)

Finally, be aware that if the activity timeout is longer than the state lifetime, then the expiration of the activity timeout also triggers the deletion of state information.

State Lifetime

The state lifetime timer determines how long state information is retained on a Store in the absence of the source or receiver. I.e. if a publisher exits, the state and message data is retained for the state lifetime period, and is then discarded.

After registration, the Store expects to hear some kind of activity (message, control, or keepalive) before the state lifetime timer expires. If not, then the Store deletes the state information associated with the source or receiver.

Setting the state lifetime is somewhat of a balancing act. If you set the source state lifetime too long, it can lead to old, stale data being available to subscribers during periods that you don't want it. If you set it too short, it risks the Store timing out the application too soon, and potentially leading to undesired message loss.

For short-lived publishers that start, register, perform some function, and exit, a fairly short state lifetime can make sense. For long-lived publishers that might have long-lasting outages and it's important for all published messages to be reliably delivered, a long state lifetime is more appropriate.

The state lifetimes default to 0, meaning that an application's state will be deleted immediately after the activity timeout happens. Most UM users set this option to a non-zero value, according to their requirements. The state lifetime can be configured by the application using: **ume_state_lifetime (source)** and **ume_state_lifetime (receiver)**. They can also be configured by the Store using: Topic Option "source-state-lifetime" and Topic Option "receiver-state-lifetime". (If both the application and the Store configures the same timer, the result varies and is described in the above linked documentation.)

Activity and State Lifetime Timers Together
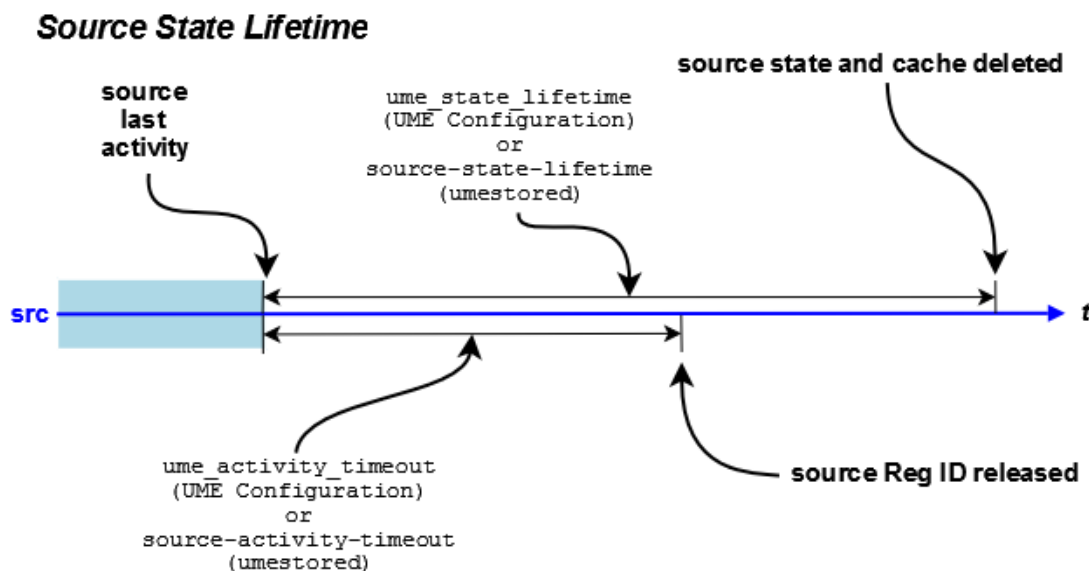
Proxy Sources

If you have enabled the Proxy Source option, a source activity timeout triggers the creation of the proxy source. The following diagram illustrates this behavior:

Source Activity and State Timers with Proxy Source
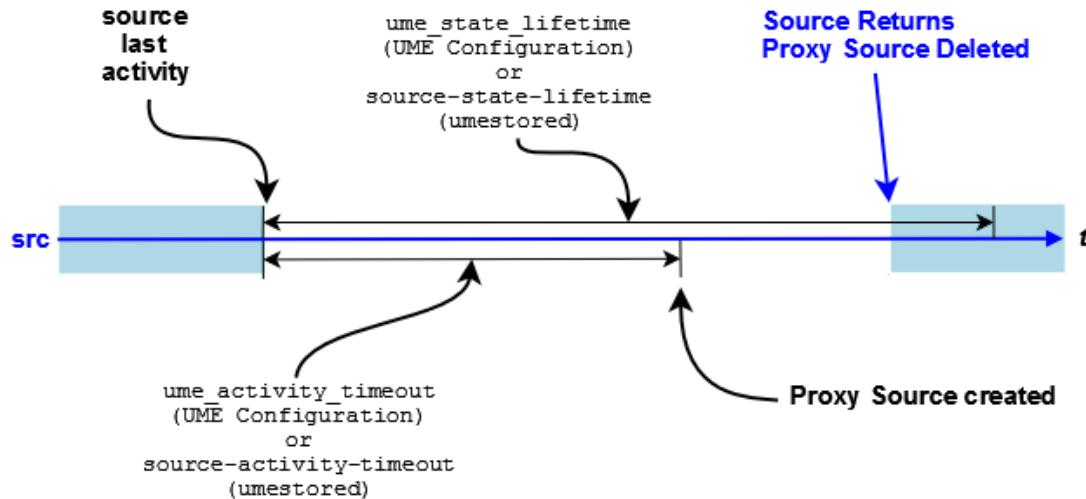
16.2.3   Enabling the Proxy Sources

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

- Configure the source in an LBM Configuration File with the source configuration option, **ume_proxy_source (source)**.

- Configure the Stores in the Store configuration file with the Store Element Option, allow-proxy-source.

16.2.4   Proxy Source Elections

When the Stores configured for proxy source detect the loss of a registered source (expiration of the source's **ume↩ _activity_timeout (source)**), one of the Stores should create a proxy source. The Stores of a Q/C group perform an election to determine which Store creates the proxy.

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 QC group or configuration of groups keeps its proxy source.

Note that Topic Option "source-activity-timeout" value should be set to at least double the Topic Option "keepalive-interval" value.

There are two algorithms that the Stores can use when holding a proxy source election:

1. Quorum not required (default),

2. Quorum required (new as of UM version 6.15; set Store Option "proxy-source-repo-quorum-required" to 1).

Informatica recommends that new projects use algorithm 2 (Quorum required). This is not the default and must be explicitly set. Existing projects that use algorithm 1 and do not have problems related to proxy sources do not need to change.

ALGORITHM DETAILS:

A proxy source is specific to a topic/reg-ID (or topic/session-ID). When a source exits (publisher deletes it or crashes), the Stores time the source out and hold an election to determine which Store will create a proxy source.

With algorithm 1 (quorum not required), every running Store in the Q/C group participates in the election.

With algorithm 2 (quorum required), only those Stores that have state for the topic/reg-ID will participate. A proxy source will be elected only if a quorum of Stores participate.

Algorithm 2 was introduced in UM version 6.15 to help customers who need to perform an un-recommended Store restart procedure whereby the state and cache files are deleted before restarting. Informatica recommends retaining the state and cache files over a restart, but we also understand that sometimes it is unavoidable and a Store must be started "clean" (for example, if a disk fails).

Creating a proxy source for a particular topic/reg-ID that does not have a quorum of repositories is contrary to the general design of UM persistence. Selecting algorithm 2 conforms with the UM persistence design.

16.2.5 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 17

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.

17.1   Source Considerations

Performance of sources is heavily impacted by:

- The Retention 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.

17.2   Receiver Considerations

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

17.2.1   Receiver 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.

17.2.2   Controlling Retransmission

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.

17.2.3   Receiver 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.

17.3   Store Configuration Considerations

Stores have numerous configuration options. See Configuration Reference for Umestored for details.

17.3.1   Configuring Store Usage per Source

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

17.3.2   Memory Use by Stores

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 Store instances within a single Store Process. This gives great flexibility on a per source basis for spreading data reception and persistent data load.

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.

17.3.3   Activity Timeouts

Stores are NOT archives and are not designed for archival. Stores persist source and receiver state with the aim of providing message recovery in the event of a fault. 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.

17.3.4   Recommendations for Store Configuration

- Number of Stores in the QC group. Informatica recommends a minimum of 3 Stores. A publisher defines the QC Store QC group using the LBM configuration option **ume_store (source)**. This option is specified multiple times to define the desired number of Stores in the QC group.

- Flight Size - Maximum number of messages sent but not stable in a quorum of Stores. The publishing application should not exceed the flight size. See Persistence Flight Size for configuration details.

- **Off-Transport Recovery (OTR)**. Informatica recommends that Stores be configured to use OTR to recover lost messages from the Source. Note that the default for **use_otr (receiver)** is "2", which does NOT enable OTR for the Store. Informatica recommends setting "use_otr" to 1 in the Store's LBM configuration file.

- Proactive Retransmissions. Informatica recommends that persistent sources use proactive retransmission to ensure message stability. See **ume_message_stability_timeout (source)** (on by default).

- **Burst Loss**. Informatica strongly recommends disabling "burst loss" by setting the LBM configuration option **delivery_control_maximum_burst_loss (receiver)** to a very large number, perhaps 10000000. This should be done for both the Store's LBM configuration and for the subscriber's LBM configuration.

- Persistence Buffer Sizes. Informatica recommends performing an analysis of expected publisher data rates and worst-case data repair times to properly size the Store's retention buffer and the source's retention buffer (late join buffer). See Persistence Buffer Sizes.

17.3.5 Store Configuration Practices to Avoid

Informatica recommends against the following Store configuration practices:

- **Burst Loss** must be disabled in Stores and persistent receivers. Set **delivery_control_maximum_burst_↩ loss (receiver)** to a very large number (1000000).

- Multiple Store QC groups require special attention. Please `contact UM Support` before using.

Chapter 18

Man Pages for Store

Persistent Store services are provided by Store Process.

There are two executables for the Store, each with it's own man page:

- Umestored Man Page - Unix and Windows command-line interface.

- Umestoreds Man Page - Windows Service interface.

18.1   Umestored Man Page

Unix and Windows command-line interface.

```
UMP Store daemon
Usage: umestored [options] configfile
Available options:
  -h, --help                   display this help and exit
  -d, --dump-dtd               dump DTD to stdout
  -f, --detach                 detach from terminal (not supported on Windows)
  -v, --validate               validate config, but do not run
  -u, --use-utc                Use UTC time format for log messages
  -a, --affinitize=PCPU[,RRLIST] assign CPU process affinity to PCPU. If optional
      RRLIST is given

                               assigns CPU affinity in a round-robin sequence as
                                  key processing
                               threads are created, example: -a 1,3,5 assigns CPU
                                  1 to the process
                               and CPUs 3, 5, 3, ... to created threads in a
                                  repeating sequence
```

Description

The `umestored` command (the final "d" stands for "daemon") runs the Store Process. It can be run inter-
actively from a shell or command prompt, or from a script or batch file. (For use as a Windows Service, see
Umestoreds Man Page.)

The **"configfile"** parameter is required and specifies the file path for the Store configuration file. See Configu-
ration Reference for Umestored for configuration details.

The **"-f"** option directs a Unix-based `umestored` to fork a child process which detaches from the controlling terminal. The `umestored` command normally remains attached to the controlling terminal and runs until interrupted. With **"-f"**, the `umestored` command exits back to the shell, and the forked child continues running in the background.

The **"-a"** option provides the CPU core affinity for Store threads. This "pins" the threads to one or more desired CPU cores, which can provide a significant improvement in throughput. See Store Thread Affinity for details.

The **"-d"** option dumps (prints) the Store's XML DTD to standard output. After dumping the DTD, `umestored` exits.

The **"-u"** option tells the Store to format its log file timestamps as UTC.

The **"-v"** option validates the XML structure of the given configuration file against the Store's XML DTD. After validating the configuration file's XML structure, `umestored` exits with status 0 for no errors, or non-zero if errors were found. For example:

```
umestored -v /um/store1_cfg.xml
```

Note that valid XML structure does not guarantee that the configuration file is completely correct. It must be tested on a running Store.

The **"-h"** option prints the man page and exits.

Exit Status

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

Usage Notes

When shutting down a Unix-based UM Persistent Store process, 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).

18.2   Umestoreds Man Page

Windows Service interface. See **UM Daemons as Windows Services** for general information about UM daemons as Windows Services.

```
UMP Store service
Usage: umestoreds [options] [configfile]
Available options:
  -E, --env_var_file          update/set environment variable file
  -U, --unset_env_var_file    unset the environment variable file
  -h, --help                  display this help and exit
  -d, --dump-dtd              dump DTD to stdout
  -u, --use-utc               Use UTC time format for log messages
  -s, --service=install       install the service passing configfile
  -s, --service=remove        delete/remove the service
```

```
-s, --service=config            update configfile info to use configfile passed
-v, --validate                  validate config, but do not run
-e, --event-log-level           update/set service logging level.  This is the
   minimum logging

                                level to send to the Windows event log.  Valid
                                  values are:
                                  NONE - Send no events
                                  INFO
                                  WARN - default
                                  ERROR
-a, --affinitize=PCPU[,RRLIST]  assign CPU process affinity to PCPU. If optional
   RRLIST is given

                                assigns CPU affinity in a round-robin sequence as
                                  key processing
                                threads are created, example: -a 1,3,5 assigns CPU
                                  1 to the process
                                and CPUs 3, 5, 3, ... to created threads in a
                                  repeating sequence
configfile                      XML config file (if not present, looks in registry)
```

Description

The `umestoreds` command has two functions:

- First, it lets the user supply Windows Service operating parameters, which the command saves into the Windows registry. Those operating parameters are subsequently used by the Store Service. See **Configure the Windows Service**.
- Second, it provides Windows with the Store Process executable to run as a Service.

The **"configfile"** parameter provides the file path for the Store configuration file. It is supplied in conjunction with the **"-v"** option or the **"-s config"** option (see below). See Configuration Reference for Umestored for configuration details.

For **"-s install"** see **Install the Windows Service**.

For **"-s remove"** see **Remove the Windows Service**.

For **"-s config"**, **"-e"**, **"-E"**, and **"-U"**, see **Configure the Windows Service**.

The **"-a"** option specifies the CPU core affinity for Store threads, which is saved in the Windows registry and subsequently by the Windows Service. This "pins" the threads to one or more desired CPU cores, which can provide a significant improvement in throughput. See Store Thread Affinity for details.

The **"-d"** option dumps (prints) the Store's XML DTD to standard output. After dumping the DTD, `umestoreds` exits.

The **"-v"** option validates the XML structure of the given configuration file against the Store's XML DTD. After validating the configuration file's XML structure, `umestoreds` exits with status 0 for no errors, or non-zero if errors were found. For example:

```
umestoreds -v c:\um\store1_cfg.xml
```

Note that valid XML structure does not guarantee that the configuration file is completely correct. It must be tested on a running Store.

The **"-h"** option prints the man page and exits.

Exit Status

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

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.

Attention

Do not use the task manager or the "kill" command to stop a UM daemon running as a Windows service. Use the Windows service control panel to stop the service. In particular, if the persistent Store is killed non-gracefully, it can leave its files in an inconsistent state.

Chapter 19

Configuration Reference for Umestored

The operating parameters for `umestored` come from a Store configuration file that must be supplied on the command line (see Man Pages for Store). A Store Process contains a UM context and receivers that may be configured with default values through an LBM configuration file referenced in the XML configuration file. Default UM options my be overridden for each configured Store using the Store configuration file.

An overview of the file format can be seen in the umestored Configuration DTD.

You configure `umestored` to instantiate Stores with the Store configuration file, which Ultra Messaging reads at start up.

The Store configuration file for persistence has the following sections:

- Daemon section - holds administrative parameters for such things as the location of log files, the LBM 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>
```

19.1   Share/Merge Store XML Files with XInclude

The XInclude mechanism can be used to merge or share XML files for UM library configuration, Store configuration, and DRO configuration. This is typically done to avoid duplicating groups of configuration options in multiple places.

To include an external file from a Store configuration file, use the following syntax:

```
<xi:include xmlns:xi="http://www.w3.org/2003/XInclude" href="FILEPATH" />
```

Where FILEPATH can be a local file name, or a network path starting with "http:" or "ftp:". For example:

```
<xi:include xmlns:xi="http://www.w3.org/2003/XInclude" href="/um/conf/TRD1.xml" />
<xi:include xmlns:xi="http://www.w3.org/2003/XInclude" href="http://myweb.mydomain.com/umconf/TRD1.xml" />
<xi:include xmlns:xi="http://www.w3.org/2003/XInclude" href="ftp://myftp.mydomain.com/umconf/TRD1.xml" />
```

Note that secure forms of network paths ("https:" or "sftp:") are not supported.

Files to be included must be formatted such that all elements are enclosed in a single container element.

Example of an **invalid** file:

```
  <option type="store" name="repository-type" value="disk"/>
  <option type="store" name="repository-size-threshold" value="75,000,000"/>
  <option type="store" name="repository-size-limit" value="100,000,000"/>
...
```

Example of **valid** file:

```
<ume-attributes>
  <option type="store" name="repository-type" value="disk"/>
  <option type="store" name="repository-size-threshold" value="75,000,000"/>
  <option type="store" name="repository-size-limit" value="100,000,000"/>
...
</ume-attributes>
```

19.1.1   Common Store XInclude Use Case

Store configuration files do not support templates. It is common that groups of configuration options need to be repeated across many Store configurations.

For example consider the Store configuration file "store_conf.xml":

```
  <stores>
    <store name="store-a1" port="14570">
...
      <topics>
        <topic pattern="matching.output\.[0-9]$" type="PCRE">
          <ume-attributes>
            <option type="store" name="repository-type" value="disk"/>
            <option type="store" name="repository-size-threshold" value="75,000,000"/>
            <option type="store" name="repository-size-limit" value="100,000,000"/>
...
          </ume-attributes>
        </topic>
      </topics>
    </store>

    <store name="store-a2" port="14571">
...
      <topics>
        <topic pattern="matching.output\.[0-9]$" type="PCRE">
          <ume-attributes>
            <option type="store" name="repository-type" value="disk"/>
            <option type="store" name="repository-size-threshold" value="75,000,000"/>
            <option type="store" name="repository-size-limit" value="100,000,000"/>
...
          </ume-attributes>
        </topic>
      </topics>
    </store>
...
```

This can be a lot of repeated content for the stores running in this daemon instance.

The XInclude feature can be used to reduce duplicate content by creating a second file "store_topic_attr.xml":

```
<ume-attributes>
  <option type="store" name="repository-type" value="disk"/>
  <option type="store" name="repository-size-threshold" value="75,000,000"/>
  <option type="store" name="repository-size-limit" value="100,000,000"/>
...
</ume-attributes>
```

Now "store_conf.xml" can be coded as:

```
  <stores>
    <store name="store-a1" port="14570">
...
      <topics>
        <topic pattern="matching.output\.[0-9]$" type="PCRE">
          <xi:include xmlns:xi="http://www.w3.org/2001/XInclude" href="./store_topic_attr.xml" />
        </topic>
      </topics>
    </store>

    <store name="store-a2" port="14571">
...
      <topics>
        <topic pattern="matching.output\.[0-9]$" type="PCRE">
          <xi:include xmlns:xi="http://www.w3.org/2001/XInclude" href="./store_topic_attr.xml" />
        </topic>
      </topics>
    </store>
...
```

19.2 Store XML Configuration File Elements

19.2.1 UMP Element "<ume-store>"

Container element that holds the configuration for the Persistent Store Process. Also defines the version of the configuration format used by the file.

- **Children:** <daemon>, <stores>

XML Attributes:

Attribute	Description	Valid Values	Default Value
version	Version number of Store XML DTD that the configuration file corresponds to. See umestored Configuration DTD for a description of the different versions. Users are encouraged to update their Store configuration files to correspond to the latest version supported by the Store software in use.	string	(no default; must be specified)

Example:

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

19.2.2 UMP Element "<stores>"

Container element for one or more <store> elements. A Store Process can run multiple independent Store instances. Some users prefer to run multiple Store instances in a single process to reduce their process management complexity. Other users prefer to run multiple Store Processes, each with a single Store instance, to reduce the impact of a Store Process failing.

There should be little or no performance difference between multiple Store instances running in the same process vs. multiple Store Processes on the same host. However, for maximum Store performance, it is generally easier to pin Store threads to cores when each process is running a single Store instance.

It is NOT recommended for multiple Stores within a QC group to run in the same process, or even the same host, as this defeats the goal of reliability through redundancy.

- **Cardinality** (number of times element can be supplied): 0 .. 1

- **Parent:** <ume-store>

- **Children:** <store>

Example:

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

19.2.3 UMP Element "<store>"

Configuration for a Store instance.

- **Cardinality** (number of times element can be supplied): 0 .. unbounded

- **Parent:** <stores>

- **Children:** <restore-last>, <publishing-interval>, <ume-attributes>, <topics>

XML Attributes:

Attribute	Description	Valid Values	Default Value
name	Identifies log messages for this Store instance in the Store Process log file, the Store Web Monitor, and Daemon Stats. Note: this is not the name that sources can use instead of a network address (see the context-name option).	attr_name	(no default; must be specified)
interface	Specifies the IP address over which Store Process 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.	string	`"0.0.0.0"` (INADDR_ANY)
port	TCP port where Store Process should listen for connection requests to this Store. Starting with UM version 6.8, zero may be supplied. In that case, the Store will choose an available port in the range **request_tcp_port_low (context)** to	string	(no default; must be specified)
	request_tcp_port_high (context). This typically requires the use of named stores.		

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      ...
    </store>
    ...
  </stores>
  ...
</ume-store>
```

19.2.4   UMP Element "<topics>"

Container for <topic> elements. Defines the topics that this Store instance will persist.

- **Parent:** <store>

- **Children:** <topic>

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      <topics>
        ...
      </topics>
      ...
    </store>
    ...
  </stores>
  ...
</ume-store>
```

19.2.5   UMP Element "<topic>"

Defines a topic pattern which the Store will use to find sources to persist. Also contains configuration information about those topics.

- **Parent:** <topics>

- **Children:** <restore-last>, <ume-attributes>

XML Attributes:

Attribute	Description	Valid Values	Default Value
pattern	A string that is used to discover sources to be persisted. The string can be a simple topic name (`type="direct"`), or it can be a regular expression (`type="pcre"`) which can match more than one topic.	string	(no default; must be specified)

Attribute	Description	Valid Values	Default Value
type	How the `pattern` attribute should be interpreted.	**`"direct"`** - Topic name (exact string match) **`"PCRE"`** - Perl regular expression. **`"regexp"`** - Posix regular expression. **Deprecated; do not use.**	**direct**

Example:

In this example, the topic "NYSE.xyz" and all topics that start with "alert." are persisted in the "MyStore1" Store instance.

```xml
<?xml version="1.0"?>
<ume-store version="1.3">
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      <topics>
        <topic pattern="NYSE.xyz" type="direct">
          ...
        </topic>
        <topic pattern="^alert\..*" type="pcre">
          ...
        </topic>
        ...
      </topics>
      ...
    </store>
    ...
  </stores>
  ...
</ume-store>
```

19.2.6 UMP Element "<ume-attributes>"

Container for a set of <option> elements.

- **Cardinality** (number of times element can be supplied): 0 .. unbounded

- **Parent:** <store>, <topic>

- **Children:** <option>

Example:

In this example, some options are at the Store level and apply to all topics. Other options are specific to the topic "NYSE.xyz".

```xml
<?xml version="1.0"?>
<ume-store version="1.3">
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      <ume-attributes>
        ...
      </ume-attributes>
      <topics>
        <topic pattern="NYSE.xyz" type="direct">
          <ume-attributes>
            ...
          </ume-attributes>
          ...
        </topic>
        ...
      </topics>
      ...
```

```
        </store>
      ...
    </stores>
    ...
</ume-store>
```

19.2.7   UMP Element "<option>"

Set a configuration option of a particular type.

This element is used to set most of the operational parameters of the Store and its repositories.

There are many different options that can be set. See "<option>" Element Details for the full list.

An option element has a type attribute. The valid types depend on whether the option's ancestor element is "<store>" or "<topic>":

Ancestor	Valid `type` Attributes
<store>	type="store" type="lbm-context"
<topic>	type="store" type="lbm-receiver" type="lbm-source"

- **Parent:** <ume-attributes>

XML Attributes:

Attribute	Description	Valid Values	Default Value
type	Type of configuration option.	**"lbm-receiver"** - LBM configuration option of scope "receiver". **"lbm-context"** - LBM configuration option of scope "context". **"lbm-source"** - LBM configuration option of scope "source". **"store"** - Store configuration option.	(no default; must be specified)
name	Name of option.	attr_name	(no default; must be specified)
value	Value for option.	string	(no default; must be specified)

Example:

```xml
<?xml version="1.0"?>
<ume-store version="1.3">
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      <ume-attributes>
        <option type="..." name="..." value="..."/>
        ...
      </ume-attributes>
      <topics>
        <topic pattern="NYSE.xyz" type="direct">
          <ume-attributes>
            <option type="..." name="..." value="..."/>
            ...
          </ume-attributes>
```

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

19.2.8   UMP Element "<restore-last>"

Control how much saved message data is restored when a disk-based Store is restarted.

By default, when a disk-based Store is stopped and restarted, it will restore all messages in the cache file to rebuild its internal index. If the cache file is very large, this can take significant time.

Using this element, you can direct the Store to read only recent data. Note that it can be specific to a Store, or a topic within a Store.

If this element is supplied in both the "<store>" and "<topic>" levels, the "<topic>" setting will override the "<store>" setting.

See Limit Initial Restore with Restore-Last for use case.

- **Cardinality** (number of times element can be supplied): 0 .. 1

- **Parent:** <store>, <topic>

XML Attributes:

Attribute	Description	Valid Values	Default Value
behavior	Define how the *value* attribute is interpreted.	**"hours"** - Directs the Store to restore the most recent *value* number of hours worth of message data. **NOTE:** the message time is relative to the last (most recent) message in the cache, *not* the absolute time that the Store is restarted. For example, if the store is restarted on Sunday, but the most-recent message was sent the previous Friday afternoon, then *value* hours worth of messages sent on Friday will be restored. **"none"** - Directs the Store to ignore *value*, disabling the feature, resulting in the entire cache file being restored.	**hours**
value	Number of units worth of message data to restore. The units are defined by the "behavior" attribute. For units of "hours", the valid range is 0 - 336 (14 days). The special value "0" disables the feature, resulting in the entire cache file being restored.	string	**0** (disable)

Example: on restart, only restore the most recent 8 hours worth of messages.

```
<?xml version="1.0"?>
<ume-store version="1.3">
...
  <stores>
    <store name="test-store" port="14567">
      <restore-last value="8" behavior="hours"/>
      ...
    </store>
  </stores>
</ume-store>
```

19.2.9   UMP Element "⟨publishing-interval⟩"

DEPRECATED: Set how often the Store publishes its Daemon Stats. See **daemonstatistics** for general information on Daemon Statistics.

Informatica requests users to migrate to using the UM configuration file to enable automatic monitoring with Protocol Buffer monitoring format for Store and DRO by setting **monitor_format (context)** to "pb". See **Automatic Monitoring**.

- **Cardinality** (number of times element can be supplied): 0 .. 1

- **Parent:** ⟨store⟩, ⟨daemon-monitor⟩

- **Children:** ⟨group⟩

Example:

Daemon Statistics are configured at both the daemon level and at the Store level.

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <daemon-monitor topic="bozo">
      <publishing-interval>
        ...
      </publishing-interval>
    </daemon-monitor>
  </daemon>
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      <publishing-interval>
        ...
      </publishing-interval>
      ...
    </store>
    ...
  </stores>
  ...
</ume-store>
```

19.2.10   UMP Element "⟨group⟩"

Configures the rate at which one particular grouping of Daemon Statistics messages are published. See **daemonstatistics** for general information on Daemon Statistics.

- **Parent:** ⟨publishing-interval⟩

XML Attributes:

Attribute	Description	Valid Values	Default Value
name	Name of statistics group being configured.	**"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.	(no default; must be specified)
ivl	Time, in seconds, between publishing the statistics group being configured.	string	(no default; must be specified)

Example:

Daemon Statistics are configured at both the daemon level and at the Store level.

```xml
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <daemon-monitor topic="bozo">
      <publishing-interval>
        <group name="default" ivl="10"/>
        <group name="store" ivl="20"/>
        ...
      </publishing-interval>
    </daemon-monitor>
  </daemon>
  <stores>
    <store name="MyStore1" interface="10.1.2.3" port="12000">
      <publishing-interval>
        <group name="default" ivl="10"/>
        <group name="store" ivl="20"/>
        ...
      </publishing-interval>
      ...
    </store>
    ...
  </stores>
  ...
</ume-store>
```

19.2.11 UMP Element "$\langle$daemon$\rangle$"

Container element for configuration elements that apply to the entire Store Process.

- **Parent:** $<$ume-store$>$

- **Children:** $<$log$>$, $<$uid$>$, $<$pidfile$>$, $<$gid$>$, $<$lbm-config$>$, $<$xml-config$>$, $<$lbm-license-file$>$, $<$web-monitor$>$, $<$daemon-monitor$>$

Example:

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

19.2.12 UMP Element "$<$daemon-monitor$>$"

DEPRECATED: The daemon-monitor element configures the Store Process for Store Binary Daemon Statistics.

Informatica requests users to migrate to using the UM configuration file to enable automatic monitoring with Protocol Buffer monitoring format for Store and DRO by setting **monitor_format (context)** to "pb". See **Automatic Monitoring**.

- **Parent:** $<$daemon$>$

- **Children:** $<$lbm-config$>$, $<$publishing-interval$>$, $<$remote-snapshot-request$>$, $<$remote-config-changes-request$>$

XML Attributes:

Attribute	Description	Valid Values	Default Value
topic	Topic name for used to publish daemon statistics.	string	`"umestore.monitor"`

Example:

Daemon Statistics are configured at both the daemon level and at the Store level.

```xml
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <daemon-monitor topic="bozo">
      <publishing-interval>
        ...
      </publishing-interval>
    </daemon-monitor>
  </daemon>
  ...
</ume-store>
```

19.2.13 UMP Element "$<$remote-config-changes-request$>$"

Controls if the daemon will respond to requests from monitoring applications. See **Daemon Control Requests** for general information, Store Daemon Control Requests for Store-specific requests.

Warning

> If misused, the Daemon Control Requests feature allows a user to interfere with the messaging infrastructure in potentially disruptive ways. By default, this feature is disabled. However, especially if you have enabled the UMP Element "<remote-config-changes-request>", Informatica recommends **Securing Daemon Control Requests**.

- **Parent:** <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
allow	Enables handling requests.	`"0"` - Disable request handling. `"1"` - Enable request handling.	`"0"`

Example:

Daemon Statistics are configured at both the daemon level and at the Store level.

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <daemon-monitor topic="bozo">
      <remote-config-changes-request allow="1"/>
        ...
      </publishing-interval>
    </daemon-monitor>
  </daemon>
  ...
</ume-store>
```

19.2.14 UMP Element "<remote-snapshot-request>"

Controls if the daemon will respond to requests from monitoring applications. See **Daemon Control Requests**.

- **Parent:** <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
allow	Enables handling requests.	`"0"` - Disable request handling. `"1"` - Enable request handling.	`"0"`

Example:

Daemon Statistics are configured at both the daemon level and at the Store level.

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <daemon-monitor topic="bozo">
      <remote-snapshot-request allow="1"/>
        ...
      </publishing-interval>
    </daemon-monitor>
  </daemon>
  ...
</ume-store>
```

19.2.15 UMP Element "<lbm-config>"

Pathname for LBM configuration file to be used when the Store creates UM objects (context, receivers, sources).

When used as a child element of <daemon-monitor>, configures the UM objects used for publishing Store Binary Daemon Statistics.

Note that starting with UM version 6.13, if one or more errors are discovered in the LBM configuration file, the errors are written to the log file and the Store continues running. I.e. errors in the LBM configuration file are treated as warnings. See **Configuration Error Handling** for an explanation.

- **Parent:** <daemon>, <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <lbm-config>/etc/ump/store0.cfg</lbm-config>
    ...
    <daemon-monitor topic="bozo">
      <lbm-config>/etc/ump/store0_dmon.cfg</lbm-config>
      ...
    </daemon-monitor>
  </daemon>
  ...
</ume-store>
```

19.2.16 UMP Element "<web-monitor>"

Address and port for the Store web-based monitor. Format is "Address:Port", where "Address" is either an IP address of one of the host's interfaces, or is "∗" which allows the use of any interface. See Store Web Monitor for more information.

If omitted, the web monitor is disabled.

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <web-monitor>*:8080</web-monitor>
    ...
  </daemon>
  ...
</ume-store>
```

19.2.17 UMP Element "<lbm-license-file>"

Pathname for UM license file. NOTE: starting with UM version 6.8, a license key is no longer required for Store operation. This element is retained for backwards compatibility.

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

19.2.18 UMP Element "<xml-config>"

Pathname for LBM configuration file (in XML format) to be used when the Store creates UM objects (context, receivers, sources). I.e. options that control the UM library.

See **XML Configuration Files** for general information on XML-based LBM configuration files.

See also <lbm-config>.

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml↩:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Attribute	Description	Valid Values	Default Value
application-name	Allows the user to select an "application name" for the Store Process, which can then used by the LBM XML configuration file to target a configuration to that Store using **UM Element "<application>"**. See **XML Application Names** for more information on application names.	string	**"umestored"**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <xml-config>/etc/ump/store01_um.xml</xml-config>
    ...
  </daemon>
  ...
</ume-store>
```

19.2.19 UMP Element "<gid>"

Specifies a Group ID (GID) for daemon process (if run as root).

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <gid>1234</gid>
    ...
  </daemon>
  ...
</ume-store>
```

19.2.20 UMP Element "<pidfile>"

Contains the pathname for daemon process ID (PID) file.

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <pidfile>/var/run/store01.pid</pidfile>
    ...
  </daemon>
  ...
</ume-store>
```

19.2.21   UMP Element "<uid>"

Specifies a User ID (UID) for daemon process (if run as root).

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <uid>1234</uid>
    ...
  </daemon>
  ...
</ume-store>
```

19.2.22   UMP Element "<log>"

Contains the path name of the Store log file. See Store Rolling Logs for more information.

If omitted, log messages are written to standard output.

- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
type	Where to write log messages.	**"file"** - Write log messages to a file. **"console"** - Write log messages to standard output.	**"console"**
frequency	Time-frame by which to roll the log file.	**"disable"** - Do not roll the log file based on time. **"daily"** - Roll the 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 use only. Do not use.	**"disable"**
size	Size (in MB, i.e. $2**20$, or 1,048,576) of current log file at which it is rolled. Specify 0 to disable rolling by log file size.	string	**"10"**
xml:space	Specifies how whitespace (tabs, spaces, linefeeds) are handled in the element content. See **xml:space Attribute**.	**"default"** - Trim whitespace. **"preserve"** - Retain whitespace exactly as entered.	**default**

Example:

```
<?xml version="1.0"?>
<ume-store version="1.3">
  <daemon>
    <log type="file" size="23" frequency="daily">/var/log/store01.log</log>
    ...
  </daemon>
  ...
</ume-store>
```

19.3   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>`').

To assist the user with upgrades, a more-recent version of the Store will accept an older version of the Store configuration file. For example, if your Store configuration file starts with this:

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

the Store will parse the file according to DTD 1.0. If you wish to use Store configuration settings that were not available in 1.0, you will need to upgrade to a later DTD version. Contact UM Support if you have trouble.

Here is the current DTD 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 (restore-last?, publishing-interval?, ume-attributes+, topics+)>
<!ATTLIST store name CDATA #REQUIRED>
<!ATTLIST store interface CDATA #IMPLIED>
<!ATTLIST store port CDATA #REQUIRED>
<!ELEMENT restore-last EMPTY>
<!ATTLIST restore-last behavior (hours | none) #IMPLIED>
<!ATTLIST restore-last value CDATA #REQUIRED>
<!ELEMENT topics (topic+)>
<!ELEMENT topic (restore-last?, 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">
```

19.4 Store Configuration Example

Store Process 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="2048"/>
            <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>
```

Chapter 20

"<option>" Element Details

The <option> element is the primary construct for setting Store configuration options. It always appears inside a <ume-attributes> block, which can appear in two places of your Store configuration file: the <store> element, or the <topic> element.

The <option> element is used to set three kinds of Store configuration options

- Setting LBM Configuration Options, for the underlying UM library.

- Store Options in "<store>" Element.

- Store Options in "<topic>" Element.

20.1   Setting LBM Configuration Options

The <option> element can be used to set LBM configuration options. I.e. options that control the UM library. Those options come in different scopes. For the purposes of the Store, only the "context", "receiver", and "source" scopes are used, as follows:

```
<option type="lbm-context" name="context_option_name" value="desired_value"/>
<option type="lbm-receiver" name="receiver_option_name" value="desired_value"/>
<option type="lbm-source" name="source_option_name" value="desired_value"/>
```

But the valid LBM scopes depend on the "<option>" element's ancestor:

Ancestor	Valid LBM `type` Attributes
<store>	type="lbm-context"
<topic>	type="lbm-receiver" type="lbm-source"

See the UM Configuration Guide for the full list of LBM 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 in an XML file. See **Interface Device Names and XML** for details.

Most Store deployments do not make heavy use of the <option> element to set LBM configuration parameters. Instead they use an LBM configuration file (via the <lbm-config> or the UMP Element "<xml-config>" element) for

setting most of the desired LBM configuration options. However, it is often desired to override one or more of those settings based on an individual Store, or an individual topic within a Store.

Example

In this hypothetical example, the Store loads a generic project-specific flat LBM configuration file:

```
...
<daemon>
  <lbm-config>/etc/ump/generic.cfg</lbm-config>
</daemon>
...
```

The file "generic.cfg" sets the **receiver-side LBT-RM socket buffer size** to 4 MB using:

```
context transport_lbtrm_receiver_socket_buffer 4194304
```

Since we want the Stores to avoid loss as much as possible, the receive socket buffers should be made large (32 MB):

```
...
<daemon>
  <lbm-config>/etc/ump/generic.cfg</lbm-config>
</daemon>
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="lbm-context" name="transport_lbtrm_receiver_socket_buffer" value="33554432"/>
    ...
  </ume-attributes>
  ...
</store>
```

Finally, for the specific topic "EventStream", one might want the initial NAK backoff interval to be longer than for application receivers.

```
...
<daemon>
  <lbm-config>/etc/ump/generic.cfg</lbm-config>
</daemon>
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="lbm-context" name="transport_lbtrm_receiver_socket_buffer" value="33554432"/>
    ...
  </ume-attributes>
  <topics>
    <topic pattern="EventStream" type="direct">
      <ume-attributes>
        <option type="lbm-receiver" name="transport_lbtrm_nak_initial_backoff_interval" value="500"/>
        ...
      </ume-attributes>
      ...
    </topic>
  </topics>
  ...
</store>
```

20.2 Store Options in "<store>" Element

For Store configuration options, the <option> element must appear inside the <store> element, as follows:

```
<store ...>
  <ume-attributes>
    <option type="store" name="store_option_name" value="desired_value"/>
  </ume-attributes>
</store>
```

For example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="disk-cache-directory" value="/cache"/>
    ...
  </ume-attributes>
</store>
```

These options apply to all repositories in this Store instance.

20.2.1   Store Option "disk-cache-directory"

Pathname for disk Store message cache directory. Must be between 1 and 230 characters long. It is the user's responsibility to create this directory; the Store will not do so.

Default:

```
umestored-cache
```

The Store looks for a sub-directory by that name in the current working directory of the stored process.

Example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="disk-cache-directory" value="/var/ump/cache"/>
    ...
  </ume-attributes>
</store>
```

20.2.2   Store Option "disk-state-directory"

Pathname for disk Store state directory. Must be between 1 and 230 characters long. It is the user's responsibility to create this directory; the Store will not do so.

Default

```
umestored-state
```

The Store looks for a sub-directory by that name in the current working directory of the Store Process.

Example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="disk-state-directory" value="/var/ump/state"/>
    ...
  </ume-attributes>
</store>
```

20.2.3   Store Option "allow-proxy-source"

Allows the Store to act as a proxy source in case a registered source terminates.

Default:

```
0 (Disable)
```

Example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="allow-proxy-source" value="1"/>
    ...
  </ume-attributes>
</store>
```

20.2.4 Store Option "proxy-source-repo-quorum-required"

Modifies the Store's proxy source election algorithm to better conform to the general UM persistence design.

See Proxy Source Elections for details.

Default:

```
0 (Disable)
```

Example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="allow-proxy-source" value="1"/>
    <option type="store" name="proxy-source-repo-quorum-required" value="1"/>
    ...
  </ume-attributes>
</store>
```

20.2.5 Store Option "context-name"

Name of the Store that can be used by sources to refer to the Store instead of the address: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 context names, which can facilitate persistent operation across the **DRO**. Store context names must be unique across the entire network.

See also Identifying Persistent Stores

Default:

None (Store must be referenced by address:port).

Example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="context-name" value="NEWYORK-1"/>
    ...
  </ume-attributes>
</store>
```

20.2.6 Store Option "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.

Default:

```
262144
```

Example:

```
<store name="MyStore1" interface="10.1.2.3" port="12000">
  <ume-attributes>
    <option type="store" name="retransmission-request-processing-rate"
                          value="524288"/>
    ...
  </ume-attributes>
</store>
```

20.3   Store Options in "<topic>" Element

For Store repository configuration options, the <option> element must appear inside the <topic> element, as follows:

```
<topic ...>
  <ume-attributes>
    <option type="store" name="store_option_name" value="desired_value"/>
  </ume-attributes>
</topic>
```

For example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-type" value="disk"/>
    ...
```

These options apply all repositories for sources that match the specified topic(s).

20.3.1   Topic Option "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. (This option should not be enabled if you anticipate using the Request: Mark Stored Message Invalid feature.)

Default:

```
0 (Store services retransmission requests and does not forward requests)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="retransmission-request-forwarding" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.2 Topic Option "repository-type"

Specifies how messages should be retained by the Store.

Possible Values

- **"memory"** retains messages only in the (presumably volatile) main memory of the Store.

- **"disk"** retains messages to disk storage. In addition, messages are cached in main memory for a time as well.

- **"reduced-fd"** DEPRECATED, do not use. Retains messages to disk storage in a lower-performing way (compared to **"disk"**) that uses fewer OS File Descriptors. This type should not be used, as it will be removed in a future UM version. Contact UM Support to devise a plan to migrate away from its use.

- **"no-cache"** DEPRECATED, do not use. Does not retain messages, only state information. This type should not be used, as it will be removed in a future UM version. Contact UM Support to devise a plan to migrate away from its use.

Default:

memory

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-type" value="disk"/>
    ...
  </ume-attributes>
</topic>
```

20.3.3 Topic Option "repository-size-threshold"

Specifies the minimum number of message bytes retained in the Store's memory cache. The purpose of this option is for disk Stores to keep some messages in memory even after they have been written to disk. This allows for rapid recovery of recent messages.

For SPP Stores, includes message payload, headers, and Store structure overhead. For RPP Stores, only includes message payload.

Note that the Store's memory cache size can fall below this threshold. With RPP Stores, if all required receivers have acknowledged consumption of all messages, all messages will be deleted.

Also for RPP, the source LBM configuration may override this option with **ume_repository_size_threshold (source)**. In that case, the Store's repository-size-threshold value is used as the maximum allowed value for ume↩ _repository_size_threshold. If the source exceeds this value, its registration is rejected.

For RPP, see RPP Configuration Specifics for interactions between this and other configuration options. See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

1024

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-size-threshold" value="2048"/>
    ...
  </ume-attributes>
</topic>
```

20.3.4 Topic Option "repository-size-limit"

Specifies the maximum number of message bytes retained in the Store's memory cache. For repository-type **"memory"**, this represents the maximum size for the entire repository. For repository-type **"disk"**, the total repository size is limited by Topic Option "repository-disk-file-size-limit".

Note that the design of UM's persistence allows a maximum of 2,147,483,647 messages ($2**31 - 1$) to be persisted. Do not specify a limit that would allow more than 2,147,483,647 messages to be stored.

For SPP Stores, includes message payload, headers, and Store structure overhead. For RPP Stores, only includes message payload.

Also for RPP, the source LBM configuration may override this option with **ume_repository_size_limit (source)**. In that case, the Store's repository-size-limit value is used the maximum allowed value for ume_repository_size_limit. If the source exceeds this value, its registration is rejected.

For RPP, see RPP Configuration Specifics for interactions between this and other configuration options. See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

50331648 (48 MB)

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-size-limit" value="67108864"/>
    ...
  </ume-attributes>
</topic>
```

20.3.5 Topic Option "repository-age-threshold"

Specifies how long in seconds the repository keeps a message available. Pertains to a memory Store or the memory cache of a disk Store. The repository reclaims space used to store messages that exceed this threshold. Note that if these deleted messages have been persisted to disk, they are available to receivers for recovery.

This is a rarely-used option, typically only useful on a memory-only Store.

A value of 0 means message age is not considered in retention decisions.

Default:

0 (disabled)

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-age-threshold" value="120"/>
    ...
  </ume-attributes>
</topic>
```

20.3.6 Topic Option "repository-disk-max-async-cbs"

This option is identical to Topic Option "repository-disk-max-read-async-cbs" and sets the same underlying limit to outstanding async reads.

The original intent of this option was to simultaneously set both Topic Option "repository-disk-max-read-async-cbs" and Topic Option "repository-disk-max-write-async-cbs". But Informatica subsequently determined that the async write limit needed to be set to 1. So this option was changed to update only the async read limit.

Default:

```
10,000 (callbacks)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-max-async-cbs" value="15000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.7 Topic Option "repository-disk-max-write-async-cbs"

For topics with a repository-type **"disk"**, specifies the maximum number of outstanding async I/O callbacks for writing messages to disk. This option is deprecated, and if supplied, must be set equal to 1. (If supplied with any other value, an error will be logged and the value will be ignored.)

Default:

```
1 (callback)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-max-write-async-cbs" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.8 Topic Option "repository-disk-max-read-async-cbs"

For topics with a repository-type **"disk"**, specifies the maximum number of outstanding async I/O callbacks for reading messages from disk. A low value can lead to severely slower message recovery rates by receivers.

Default:

```
10,000 (callbacks)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-max-read-async-cbs" value="15000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.9 Topic Option "repository-disk-file-size-limit"

Specifies the maximum amount of disk space, in bytes, that will be used to store retained messages. A minimum value of 196992 is enforced.

Note that the design of UM's persistence allows a maximum of 2,147,483,647 messages ($2**31$ - 1) to be persisted. Do not specify a limit that would allow more than 2,147,483,647 messages to be stored.

This option only applies for repository-type **"disk"**.

Also for RPP, the source LBM configuration may override this option with **ume_repository_disk_file_size_limit (source)**. In that case, the Store's repository-disk-file-size-limit value is used as the maximum allowed value for ume_repository_disk_file_size_limit. If the source exceeds this value, its registration is rejected.

For RPP, see RPP Configuration Specifics for interactions between this and other configuration options. See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

```
104857600 (100 MB)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-file-size-limit" value="209715200"/>
    ...
  </ume-attributes>
</topic>
```

20.3.10 Topic Option "repository-disk-file-preallocate"

For topics with a repository-type **"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.

Default:

```
0 (do not pre-allocate)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-file-preallocate" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.11 Topic Option "repository-disk-async-buffer-length"

For topics with a repository-type **"disk"**, 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.

Default:

```
1024000 (bytes)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-async-buffer-length" value="2097152"/>
    ...
  </ume-attributes>
</topic>
```

20.3.12   Topic Option "repository-disk-message-checksum"

For topics with a repository-type **"disk"**, specifies whether the messages saved to disk should include a checksum field for validation if the Store is restarted.

Default:

```
0 (disabled)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-message-checksum" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.13   Topic Option "source-activity-timeout"

Establishes the period of time in milliseconds 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.

Note that Topic Option "source-activity-timeout" value should be set to at least double the Topic Option "keepalive-interval" value.

Default:

```
30000 (30 seconds)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="source-activity-timeout" value="60000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.14 Topic Option "source-state-lifetime"

Establishes the period of time in milliseconds 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.

Default:

```
0 (disabled)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="source-state-lifetime" value="0"/>
    ...
  </ume-attributes>
</topic>
```

20.3.15 Topic Option "receiver-activity-timeout"

Establishes the period of time in milliseconds from a receiver's last activity to the release of the receiver's Reg↩ ID. Stores return an error to any new request for the receiver's RegID during this period. Can be overridden by **ume_activity_timeout (receiver)**. See also Persistence Proxy Sources.

Default:

```
30000 (30 seconds)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="receiver-activity-timeout" value="45000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.16 Topic Option "receiver-state-lifetime"

Establishes the period of time in milliseconds 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.

Default:

```
0 (disabled)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="receiver-state-lifetime" value="0"/>
    ...
  </ume-attributes>
</topic>
```

20.3.17 Topic Option "source-check-interval"

Specifies the period in milliseconds a Store will check for activity of sources and receivers.

Default:

750 (milliseconds)

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="source-check-interval" value="900"/>
    ...
  </ume-attributes>
</topic>
```

20.3.18 Topic Option "keepalive-interval"

Specifies the period in milliseconds a Store will generate keepalive traffic to sources and receivers if there has been no traffic required in the normal course of operation.

Note that Topic Option "source-activity-timeout" value should be set to at least double the Topic Option "keepalive-interval" value.

Default:

3000 (3 seconds)

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="keepalive-interval" value="5000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.19 Topic Option "receiver-new-registration-rollback"

Specifies an upper limit to the number of messages a registering persistent receiver should recover during "late join".

Attention

> For most use cases, this option should be left at its default, which effectively disables a limit.

The receiver can limit the number of messages that it can request for recovery using the LBM configuration option **retransmit_request_maximum (receiver)**. If the request exceeds the limit configured in the Store's topic option receiver-new-registration-rollback, the receiver's recovery will be limited to the latter number of messages.

Note that this limit interferes with the persistence guarantee of delivery. If a limit of 1,000 is configured, and a receiver exits and restarts after the source has sent 4,000 more messages, the receiver might not be able to recover all 4,000 messages. It might be limited to the most-recent 1,000 messages.

Also, note that a persistent receiver can set up a initial sequence number callback using the LBM configuration option **ume_recovery_sequence_number_info_function (receiver)**. This allows the application to specify a desired

starting sequence number. If the application is written to use this approach, Store topic option "receiver-new-registration-rollback" and LBM configuration option retransmit_request_maximum should be at their default values.

If used, the value for receiver-new-registration-rollback must be between 1 and 2147483647 (maximum signed 32-bit integer). The default value of 2147483647 effectively disables the limit.

Default:

```
2147483647
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="receiver-new-registration-rollback" value="2147483647"/>
    ...
  </ume-attributes>
</topic>
```

20.3.20   Topic Option "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.

Default:

```
60,000 (60 seconds)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="proxy-election-interval" value="80000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.21   Topic Option "stability-ack-interval"

Specifies the maximum amount of time in milliseconds that stability acknowledgments will be batched before being sent to a source. This batching is only enabled if Topic Option "stability-ack-minimum-number" is set to greater than 1.

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.

At high message rates, the stability ACKs will normally be triggered by the received messages exceeding stability-ack-minimum-number. However, if the source publishing rate drops significantly, the stability-ack-interval ensures an upper bound to the time required for stability ACKs.

Default:

```
200 (200 milliseconds)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="stability-ack-interval" value="50"/>
    ...
  </ume-attributes>
</topic>
```

20.3.22   Topic Option "stability-ack-minimum-number"

Specifies the minimum number of message fragment 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 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.

For RPP, see RPP Configuration Specifics for interactions between this and other configuration options. See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

```
1 (fragment; no batching)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="stability-ack-minimum-number" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.23   Topic Option "repository-allow-receiver-paced-persistence"

Specifies if the repository allows receiver-paced persistence (RPP). If allowed (value 1), the source may request RPP with **ume_receiver_paced_persistence (source)**.

Note that the Store cannot be directly configured to enable RPP; the source *must* be configured to request it. Otherwise, the repository defaults to SPP. This option only *allows* the source to request it.

See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

```
0 (Store does not allow the source to specify RPP)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-allow-receiver-paced-persistence" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.24 Topic Option "repository-allow-ack-on-reception"

For RPP, specifies if the repository allows "ack on reception" behavior. If allowed (value 1), the source may request "ack on reception" with **ume_repository_ack_on_reception (source)**. See **ume_repository_ack_on_reception (source)** for more information about "ack on reception" behavior.

Note that the Store cannot be directly configured to enable "ack on reception"; the source *must* be configured to request it. Otherwise, the repository defaults to acknowledging when messages are written to disk. This option only *allows* the source to request it.

Also note that for SPP Stores, this option is ignored.

For RPP, see RPP Configuration Specifics for interactions between this and other configuration options. See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

0 (Store does not allow the source to specify ack on reception behavior)

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-allow-ack-on-reception" value="1"/>
    ...
  </ume-attributes>
</topic>
```

20.3.25 Topic Option "repository-disk-write-delay"

Specifies a delay in milliseconds after a message is received before the message is written to disk. This option is for use with RPP, and is intended to allow all required receivers to acknowledge consumption of messages within the write delay time. This deletes the messages from the memory cache before they are written to disk. If all required receivers acknowledge consumption within the delay time, the Store never needs to write messages to disk.

This option only applies for repository-type **"disk"**.

For RPP, the source LBM configuration may override this option with **ume_write_delay (source)**. In that case, the Store's repository-disk-write-delay value is used as the maximum allowed value for ume_write_delay. If the source exceeds this value, its registration is rejected.

For RPP, see RPP Configuration Specifics for interactions between this and other configuration options. See RPP: Receiver-Paced Persistence for general information on RPP.

Default:

0 (milliseconds, no delay)

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="repository-disk-write-delay" value="5000"/>
    ...
  </ume-attributes>
</topic>
```

20.3.26 Topic Option "source-flight-size-bytes-maximum"

For 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 exceeds this value, its registration is rejected.

See Persistence Flight Size for more information.

Default:

```
4194304 (4 MB)
```

Example:

```
<topic pattern="EventStream" type="direct">
  <ume-attributes>
    <option type="store" name="source-flight-size-bytes-maximum" value="8388608"/>
    ...
  </ume-attributes>
</topic>
```

Chapter 21

Special Configuration Topics

21.1 Store Loss Repair

The persistent Store uses a normal UM receiver to get messages from the source. The Store is subject to the same potential **Packet Loss** scenarios as an application subscriber, and uses the same loss repair techniques as a subscriber.

Informatica recommends enabling **Off-Transport Recovery (OTR)** on the Store and application subscribers.

21.2 Persistence Buffer Sizes

There are two memory buffers that need to be properly sized to get the desired level of performance and reliability for persistence:

- Source retention buffer, also known as the late join buffer.

- Store message cache.

This analysis assumes that you are using disk-based Stores.

The sizing depends on how sensitive your publishing application is to being slowed down in the event of packet loss. The Stores are designed to write messages to disk in sequence-number order. So in the event of packet loss, newly received messages must be buffered in the Store's message cache while the Store waits for retransmission of the lost messages. If the source's message rate is high and the time required to repair the packet loss is significant, the Store's cache must be configured to be large.

Note that this affects the flight size configuration also. For SPP Stores, message stability is acknowledged to the source after the messages is successfully written to disk. But for messages being kept in the Store's message cache waiting for lost packet retransmission, stability acknowledgement is delayed. All messages sent during this time are said to be "in flight". To avoid the publisher being blocked on flight size, the flight size limit might need to be made large. It is not unusual for our performance-sensitive users to configure the flight size limit to be in the tens of thousands.

Finally, the source's retention buffer (late join buffer) should be sized the same as the flight size limit.

21.3 Calculating Options for SPP

Determine the following for your application:

- avg_msg_size - The source's average size of application messages, in bytes.

- avg_msg_rate - The source's average message send rate, in datagrams per second.

The following formulas calculate minimum recommended values of some configuration options:

```
ume_flight_size = 3 * (ume_ack_batching_interval/1,000) * avg_msg_rate
ume_flight_size_bytes = ume_flight_size * avg_msg_size
ume_repository_size_threshold = ume_flight_size_bytes
ume_repository_size_limit = 1.2 * ume_repository_size_threshold
```

If Topic Option "stability-ack-minimum-number" is greater than 1, the value of Topic Option "stability-ack-interval" needs to be added in as follows:

```
ume_flight_size = 3 * ((ume_ack_batching_interval + stability-ack-interval)/1,000)
                * avg_msg_rate
```

The other parameters must then be recalculated.

For example:

```
avg_msg_size = 1024 bytes            (hypothetical use case)
avg_msg_rate = 10,000 datagrams/sec  (hypothetical use case)

ume_flight_size = 3 * (100/1000) * 10000 = 3000
ume_flight_size_bytes = 3000 * 1024 = 3,072,000
ume_repository_size_threshold = 3,072,000
ume_repository_size_limit = 1.2 * 3072000 = 3,686,400
```

Note that application deviations (e.g. bursts) from the averages can result in unexpected disk writes and blocking imposed on the source. To reduce the chances of blocking, the values for avg_msg_size and/or avg_msg_rate can be increased (with the subsequent configuration values recalculated).

21.4 RPP Configuration Specifics

With RPP, there is a non-obvious interaction between the settings for:

- Store configuration option repository-size-limit,

- Store configuration option repository-size-threshold

- Store configuration option repository-disk-write-delay,

- Source LBM configuration option **ume_repository_ack_on_reception (source)**,

- Source LBM configuration options **ume_flight_size_bytes (source)** and **ume_flight_size (source)**.

- Receiver LBM configuration option **ume_ack_batching_interval (context)**.

- Store LBM configuration options stability-ack-interval and stability-ack-minimum-number.

The source and Store can enter a state where the sending rate is severely limited by flight size, even though the Store is relatively idle.

To avoid this issue, you need to analyze your usage patterns and set your configuration options appropriately. Determine the following for your application:

- avg_msg_size - The source's average size of application messages, in bytes.

- avg_msg_rate - The source's average message send rate, in datagrams per second.

Next you need to decide how long you want to set repository-disk-write-delay. The idea is that you want to avoid writing to disk during normal operation, so you should set it long enough that all normally-operating receivers have a chance to acknowledge consumption of the messages during the write delay time. Remember that when a receiver gets a message, it might delay sending consumption for **ume_ack_batching_interval (context)** milliseconds (defaults to 100).

You also need to take into account heavy bursts of traffic, where receivers might store significant numbers of messages in their socket buffers, and it can take time for the receivers to work their way through all of them.

But you don't want to make the repository-disk-write-delay larger than necessary because it can lead to very high memory usage in the Store. We generally see repository-disk-write-delay being set to values as low as 1000 (1 sec) and as high as 5000 (5 sec).

The following formulas calculate minimum recommended values of some configuration options:

```
ume_flight_size = 3 * (ume_ack_batching_interval/1,000) * avg_msg_rate
ume_flight_size_bytes = ume_flight_size * avg_msg_size
ume_repository_size_threshold =
        (avg_msg_size * avg_msg_rate * (repository-disk-write-delay/1,000))
        + ume_flight_size_bytes
ume_repository_size_limit = 1.2 * ume_repository_size_threshold
```

If Topic Option "stability-ack-minimum-number" is greater than 1, the value of Topic Option "stability-ack-interval" needs to be added in as follows:

```
ume_flight_size = 3 * ((ume_ack_batching_interval + stability-ack-interval)/1,000)
                * avg_msg_rate
```

The other parameters must then be recalculated.

For example:

```
avg_msg_size = 1024 bytes           (hypothetical use case)
avg_msg_rate = 10,000 datagrams/sec  (hypothetical use case)
repository-disk-write-delay = 2500  (2.5 sec, chosen by user)

ume_flight_size = 3 * (100/1000) * 10000 = 3000
ume_flight_size_bytes = 3000 * 1024 = 3,072,000
ume_repository_size_threshold = (1024 * 10000 * (2500/1000)) + 3072000 = 28,672,000
ume_repository_size_limit = 1.2 * 28672000 = 34,406,400
```

Note that application deviations (e.g. bursts) from the averages can result in unexpected disk writes and blocking imposed on the source. To reduce the chances of blocking, the values for avg_msg_size and/or avg_msg_rate can be increased (with the subsequent configuration values recalculated).

Chapter 22

Store Binary Daemon Statistics

Note

> The C-style binary structure format of daemon statistics is DEPRECATED and may be removed in a future release. Informatica requests that users migrate to protobuf-based format. See **Monitoring UM Daemons**.

See **Example Protocol Files** for the protocol buffer definition files.

22.1   Store Daemon Statistics Structures

Note

> the C-style binary structure format of daemon statistics is DEPRECATED and may be removed in a future release. Informatica requests that users migrate to protobuf-based format. See **Monitoring UM Daemons**.

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.

22.1.1   Store Daemon Statistics Byte Swapping

Note

> the C-style binary structure format of daemon statistics is DEPRECATED and may be removed in a future release. Informatica requests that users migrate to protobuf-based format. See **Monitoring UM Daemons**.

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 **Example 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.

22.1.2   Store Daemon Statistics String Buffers

Note

> the C-style binary structure format of daemon statistics is DEPRECATED and may be removed in a future release. Informatica requests that users migrate to protobuf-based format. See **Monitoring UM Daemons**.

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 **DRO 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.)

22.1.3 Store Daemon Statistics Retx Counts

Note

> the C-style binary structure format of daemon statistics is DEPRECATED and may be removed in a future release. Informatica requests that users migrate to protobuf-based format. See **Monitoring UM Daemons**.

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.

22.2 Store Daemon Statistics Configuration

Note

> the C-style binary structure format of daemon statistics is DEPRECATED and may be removed in a future release. Informatica requests that users migrate to protobuf-based format. See **Monitoring UM Daemons**.

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

- The UMP Element "<daemon-monitor>" inside the UMP Element "<daemon>" configures all aspects of the Store Daemon Statistics feature, including publishing intervals, for all Store instances in a Store Process.

- The UMP Element "<publishing-interval>" inside a UMP Element "<store>" configures only the publishing intervals of a Store instance.

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.

22.3   Store Daemon Control Requests

The Store Process supports a monitoring application to send a specific set of requests to control the operation of Daemon Statistics, and other operations of the Store. The <remote-snapshot-request> and <remote-config-changes-request> elements control whether the Store enables the **Daemon Controller** operation (both default to disabled).

Warning

> If misused, the Daemon Control Requests feature allows a user to interfere with the messaging infrastructure in potentially disruptive ways. By default, this feature is disabled. However, especially if you have enabled UMP Element "<remote-config-changes-request>", Informatica recommends **Securing Daemon Control Requests**.

If enabled, the monitoring application can send a request 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 request, 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.

22.3.1   Store Daemon Control Request Addressing

Since Daemon Control Requests are sent as UIM messages, you must use a target string to address the request to the desired Store Process. The general form of a UIM target address is described in **UIM Addressing**, but is illustrated by this example:

```
TCP:10.29.3.46:12009
```

where 10.29.3.46:12009 is the IP and Port of the Daemon Control context UIM port. These are typically configured using the **request_tcp_interface (context)** and **request_tcp_port (context)** options in the LBM configuration file specified by the UMP Element "<lbm-config>" contained within the UMP Element "<daemon-monitor>".

22.3.2 Store Daemon Control Request Types

The example program **Example umedcmd.c** demonstrates the correct way to send the messages and receive the responses. See umedcmd Man Page for usage details.

REQUEST TYPES ENABLED BY <**remote-snapshot-request**>**:**

version

    The Store returns in its 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_H↩EAP_STATS**.

snap src

    The Store immediately publishes the source repository statistics message(s) **LBM_UMESTORE_DMON_M↩PG_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_ST↩ATS**.

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_ST↩ORE_CONFIG**, **LBM_UMESTORE_DMON_MPG_STORE_PATTERN_CONFIG**, **LBM_UMESTORE_DMO↩N_MPG_STORE_TOPIC_CONFIG**, **LBM_UMESTORE_DMON_MPG_REPO_CONFIG**, and **LBM_UMEST↩ORE_DMON_MPG_RCV_CONFIG**

REQUEST TYPES ENABLED BY <**remote-config-changes-request**>**:**

A Store Process can have multiple Store instances. But the UIM message is sent to the Daemon Control context within the Store Process.

Except as noted, the following requests can either be applied to all Store instances in the Store Process, or to just one Store instance. To apply the request to one Store instance, the Store name (as specified in the UMP Element "<store>" attribute "name") should be specified in double quotes.

memory *N*

> Set the publishing interval for memory usage. This is only available on a Store Process basis. A Store instance may not be supplied.
>
> For example: `memory 5`

src *N*

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

For the following requests, a Store instance *must* be supplied as part of the request. It is supplied as an IP and Port, as specified in the UMP Element "<store>" attributes "interface" and "port". Note that the following requests are *not* related to the Daemon Statistics feature, but are nonetheless enabled by <remote-config-changes-request>.

mark *INTFC PORT SRC_REGID SQN*

> Mark as invalid the message with sequence number *SQN* from the source with registration ID *SRC_REGID* on the Store instance at *INTFC:PORT*.
>
> For example: `mark 10.29.3.16 12000 127025183 500`
>
> Note that only one sequence number can be specified. See Request: Mark Stored Message Invalid for more information.

deregister *INTFC PORT SRC_REGID RCV_REGID*

> Deregister the receiver with registration ID *RCV_REGID* associated with the source with registration ID *SRC_REGID* on the Store instance at *INTFC:PORT*.
>
> For example: `deregister 10.29.3.16 12000 127025183 127025184`
>
> Note that only one receiver registration ID can be specified. See Request: Deregister Receiver for more information.

22.3.3   Request: Mark Stored Message Invalid

There are occasions when a user might want to mark one or more messages in a Store's repository as invalid, to prevent them from being delivered to a recovering receiver. This can be useful if a misbehaving publisher sends a "poison" message that causes receivers to crash; having that message in the Store's repository means that restarting the failed receiver will just cause it to crash again when the message is recovered.

This message marking feature is provided by the daemon command-and-control feature Store Daemon Control Requests. Note that if there is more than one Store instance in this QC group, the request needs to be sent multiple times, once for each Store instance IP/Port.

Warning

> If misused, the Daemon Control Requests feature allows a user to interfere with the messaging infrastructure in potentially disruptive ways. By default, this feature is disabled. However, especially if you have enabled UMP Element "<remote-config-changes-request>", Informatica recommends **Securing Daemon Control Requests**.
>
> When a message is marked invalid with this feature, that mark is *NOT* saved onto disk. If the marked message resides on disk and the Store is restarted, it loses its invalid mark and becomes subject to delivery to recovering receivers. Invalid messages may need to be re-marked as invalid after a Store restart.
>
> The message marking feature is incompatible with the retransmission-request-forwarding Store option. If you have configured the Store instance to enable the retransmission-request-forwarding option, and a recovering receiver requests a message that has been marked as invalid, the Store instance will forward the recovery request to the source. If the source still has the message in its retention buffer, the Store will supply it to the receiver.

Daemon Control requests can be sent by the example program **Example umedcmd.c**. Alternatively, that program's source code can be used as a guide for writing your own Store management program. See umedcmd Mark Mode for full details.

22.3.4   Request: Deregister Receiver

There are occasions when a user might want to deregister a failed receiver from a Store. This will delete the Store's state information for that receiver.

This receiver deregistration feature is provided by the Daemon command-and-control feature Store Daemon Control Requests.

Warning

> If misused, the Daemon Control Requests feature allows a user to interfere with the messaging infrastructure in potentially disruptive ways. By default, this feature is disabled. However, especially if you have enabled UMP Element "<remote-config-changes-request>", Informatica recommends **Securing Daemon Control Requests**.

A receiver's state information is stored per-source. For example, if an application creates a persistent receiver for topic X, and there are two sources for topic X, the Store will save two sets of state information for that receiver, one for each source for X. To fully clean up a failed receiving application, you need to deregister every pairing of receiver registration ID (RegID) associated with that receiver with every source RegID. And that must be repeated for each Store instance that the receiver was registered with. (Session IDs may not be used.)

Once deregistered, the state and cache files are deleted and cannot be restored.

Note that if there is more than one Store instance in this QC group, the request needs to be sent multiple times, once for each Store instance IP/Port.

Note

> If you use this feature to deregister a receiver that is still running, that receiving application is not informed of the deregistration, and it will continue to receive messages from the source and will attempt to acknowledge them to the Store. However, the Store will discard these acknowledgements as invalid and will log warnings to its log file. The receiving application will not be aware of the acknowledgement discards.

Daemon Control Requests can be sent by the example program **Example umedcmd.c**. Alternatively, that program's source code can be used as a guide for writing your own Store management program. See umedcmd Deregister Mode for full details.

22.4   umedcmd Man Page

The umedcmd example program sends **Daemon Control Requests** to a Store Process. Source code for umedcmd can be found with the other example programs; see **Example umedcmd.c**.

Note

> UM version 6.13 has a known issue running umedcmd on Windows. See **Known Issue 10897**.

The umedcmd command has 3 modes of operation:

- publish

    - used to control the publishing of Daemon Statistics by the Persistent Store.

- mark

    - used to mark messages in a Store instance as invalid.

- deregister

    - used to deregister and delete state information for a receiver that is currently registered with a Store instance.

Each mode has a different usage pattern, which is determined by the value passed to the "-m" command-line option.

22.4.1   umedcmd Publish Mode

This form of the umedcmd command is used to control the publishing of Daemon Statistics by the Persistent Store.

```
*****************************************************************************
Usage: umedcmd -m publish -c config_file -T target_string [-L linger]
       [command_string]
Available options:
  -c, --config=FILE       Use LBM configuration file FILE.
                             Multiple config files are allowed.
                             Example: '-c file1.cfg -c file2.cfg'
  -h, --help              display this help and exit
  -L, --linger=NUM        linger for NUM seconds before closing context
  -m, --mode=TYPE         set the command mode to TYPE 'publish' [required]
  -T, --target=TARGET     TARGET string for unicast immediate messages
                          [required]
*****************************************************************************
```

The "-m mode" command-line option is optional in this usage. If supplied, it must be supplied as "-m publish". Omitting it defaults to publish mode.

The "-T target_string" contains the unicast immediate message destination address of the Daemon Control context UIM port (see Store Daemon Control Request Addressing).

The parameter "command_string" is optional. If supplied, it should be enclosed in single quotes. If omitted, the program enters an interactive mode in which the user can enter any number of commands (when used interactively, do not enclose the command string in single quotes). In interactive mode, use "h" for a brief help screen, and "q" to quit.

Valid command strings are:

```
*****************************************************************************
* Publish Mode                                                              *
* help (print this message): h                                             *
*   quit (exit application): q                                             *
* report store dmon version: version                                       *
*   set publishing interval: memory 0-N                                     *
*                            ["store name"] src 0-N                         *
*                            ["store name"] rcv 0-N                         *
*                            ["store name"] disk 0-N                        *
*                            ["store name"] store 0-N                       *
*                            ["store name"] config 0-N                      *
*      snapshot all groups: ["store name"] snap memory|src|rcv|disk|store|config *
*****************************************************************************
```

Note that most of the commands can optionally be preceded by a Store instance name in double quotes. Supplying it causes the command to apply only to the named Store instance. Omitting this causes the command to apply to *all* Store instances in the target Store Program.

For example:

```
umedcmd -c dstats.cfg -m publish -T TCP:10.29.3.16:12009 '"store1" src 5'
```

In this example, the Store Process's Daemon Control context has its UIM port configured as 12009 (see Store Daemon Control Request Addressing), and the Store instance is configured for the name "store1". (with the UMP Element "<store>" attribute "name"). The source repository statistics are set to a publishing interval of 5 seconds.

22.4.2   umedcmd Mark Mode

This form of the umedcmd command is used to mark persisted messages as invalid. This prevents their delivery to recovering receivers. See Request: Mark Stored Message Invalid.

```
*****************************************************************************
Usage: umedcmd -m mark -c config_file -i store_interface -p store_port
      -s src_regid -T target_string [-L linger] [-S sqn_string]
Available options:
  -c, --config=FILE     Use LBM configuration file FILE.
                           Multiple config files are allowed.
                           Example: '-c file1.cfg -c file2.cfg'
  -h, --help            display this help and exit
  -i, --store_interface store interface IPv4 address [required]
  -p, --store_port      store port [required]
  -L, --linger=NUM      linger for NUM seconds before closing context
  -m, --mode=TYPE       set the command mode to TYPE 'mark' [required]
  -s, --src_regid=ID    source registration ID associated with the store
                        repository [required]
  -S, --sqn_string=LIST LIST of one or more message sequence number(s) or
                        ranges to drop], e.g.:
                           '-S 54'      drop a single message
```

```
                              '-S 312-315'  drops a range of messages
                              '-S 2,5,7-9' drops two single and a range of messages
  -T, --target=TARGET   TARGET string for unicast immediate messages
                        [required]
*******************************************************************************
```

The "-m mark" command-line option must be supplied.

The "-T target_string" contains the unicast immediate message destination address of the Daemon Control context UIM port (see Store Daemon Control Request Addressing).

The "-i store_interface" and "-p store_port" are required parameters which identify the desired Store instance within the Store Process, as specified in the UMP Element "<store>" attributes "interface" and "port".

The "-s src_regid" parameter is required to identify the specific source that sent the invalid message.

The command-line option "-S sqn_string" specifies the sequence number(s) of the messages that should be marked invalid. If omitted, the program enters an interactive mode in which the user can enter any number of sequence number strings.

A sequence number string can specify multiple sequence numbers and/or ranges of sequence numbers. A range is two sequence numbers separated by a dash. The string can consist of one or more sequence numbers or ranges, separated by commas. The string should be enclosed in quotes. For example:

```
-S "100,110-112,220"
```

This specifies sequence numbers 100, 110, 111, 112, 220. Note that the `umedcmd` command parses the sequence number string and issues a separate request to the Store instance for each sequence number.

If "-S sqn_string" is omitted from the command line, the program enters an interactive mode in which the user can enter any number of sequence number strings. In interactive mode, use "h" for a brief help screen, and "q" to quit.

For example:

```
umedcmd -c dstats.cfg -m mark -T TCP:10.29.3.16:12009 -i 10.29.3.16 -p 12000 -s
    127025183 -S "500"
```

In this example, the Store Process's Daemon Control context has its UIM port configured as 12009 (see Store Daemon Control Request Addressing), and the Store instance is configured for port 12000 (with the "port" attribute of the UMP Element "<store>"). The source registration ID is 127025183. The message with sequence number 500 is marked invalid.

Note

> If there is more than one Store instance in this QC group, the command needs to be executed multiple times, once for each Store instance IP/Port.

22.4.3 umedcmd Deregister Mode

This form of the umedcmd command is used to deregister a failed receiver. This deletes the state information for that receiver. See Request: Deregister Receiver.

```
*******************************************************************************
Usage: umedcmd  -m deregister-c config_file -i store_interface -p store_port
      -s src_regid -T target_string [-r rcvr_regid] [-L linger]
Available options:
  -c, --config=FILE       Use LBM configuration file FILE.
                             Multiple config files are allowed.
                             Example: '-c file1.cfg -c file2.cfg'
  -h, --help              display this help and exit
  -i, --store_interface   store interface IPv4 address [required]
  -p, --store_port        store port [required]
```

```
-L, --linger=NUM        linger for NUM seconds before closing context
-m, --mode=TYPE         set the command mode to TYPE 'deregister' [required]
-r, --rcvr_regid=LIST   LIST of one or more receiver registration IDs
                        with store repository, e.g.:
                         '-r 127025171'          deregister single receiver
                         '-r 127025171, 127025162' deregister two receivers
-s, --src_regid=ID      source registration ID associated with the
                        store repository [required]
-T, --target=TARGET     TARGET string for unicast immediate messages
                        [required]
****************************************************************************
```

The "-m deregister" command-line option must be supplied.

The "-T target_string" contains the unicast immediate message destination address of the Daemon Control context UIM port (see Store Daemon Control Request Addressing).

The "-i store_interface" and "-p store_port" are required parameters which identify the desired Store instance within the Store Process, as specified in the UMP Element "<store>" attributes "interface" and "port".

The "-s src_regid" and "-r rcv_regid" parameters combine to identify the specific receiver state that will be deleted. Receiver state is stored according to a *pair* of registration IDs: source, receiver. (Session IDs may not be used.) For example, lets say there are two persisted sources for the same topic with registration IDs 100 and 200. A receiver with registration ID 300 will have two sets of state: state for the pair 100, 300 and state for the pair 200, 300.

Note that the "-r rcv_regid" parameter can have a comma-separated list of receiver registration IDs. This is handy if you need to de-register all receivers for a particular source. The rcv_regid should be enclosed in quotes. Note that `umedcmd` command parses the receiver registration IDs and issues separate request to the Store instance for each rcv_regid.

Also note that if "-r rcv_regid" is omitted from the command line, the program enters an interactive mode in which the user can enter any number of receiver registration IDs. In interactive mode, use "h" for a brief help screen, and "q" to quit.

For example:

```
umedcmd -c dstats.cfg -m deregister -T TCP:10.29.3.16:12009 -i 10.29.3.16 -p 12000
    -s 127025183 -r "127025184"
```

In this example, the Store Process's Daemon Control context has its request port configured as 12009 (see Store Daemon Control Request Addressing), and the Store instance is configured for port 12000 (with the "port" attribute of the UMP Element "<store>"). The source registration ID is 127025183 and the receiver registration ID is 127025184. This pair of registration IDs is used by the Store instance to delete the receiver state.

Note

> If there is more than one Store instance in this QC group, the command needs to be executed multiple times, once for each Store instance IP/Port. Also, if there is more than one source for the same topic that the receiver is registered for, the command needs to be executed multiple times, once for each source registration ID.

Chapter 23

Store Web Monitor

Note

> The Store web monitor functionality is deprecated in favor of **MCS**. We do not plan to remove existing web monitor functionality, and will continue to support it in its current state. But we do not plan to enhance the web monitor in the future.

The built-in web monitor (configured in the Store configuration file) is a rich source of information about the health of a Store. 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.

Warning

> The Store's web monitor is not designed to be a highly-secure feature. Anybody with access to the network can access the web monitor pages.

Users are expected to prevent unauthorized access to the web monitor through normal firewalling methods. Users who are unable to limit access to a level consistent with their overall security needs should disable the Store web monitor (using <web-monitor>). See **Webmon Security** for more information.

23.1   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.

23.2   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.

23.3   Store Web Monitor Store Page

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

```
Store 0: UME_Store_sr71prod-tk_1a

        Interface: 0.0.0.0:38401
        Cache Dir: /tmp/umestore/cache_1
        State Dir: /tmp/umestore/state_1
        Configured Retransmission Request Processing Rate: 262144
        Total seconds used for rate calculations: 486
        Retransmission Request Received Rate: 5.000000
        Retransmission Request Service Rate: 5.000000
        Retransmission Request Drop Rate: 0.000000
        Retransmission Request Total Dropped: 0
        Patterns: 1
              • ".", PCRE
        Topics: 1
              • "test1" - 2504558780(39307788)
        Reset Rate Stats
```

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 option in the Store configuration file. For example: <option type="store" name="disk-cache-directory" value="cache/" />
State Dir	Pathname for disk Store state directory. This would be configured as a Store option in the Store 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.

Item	Description
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 option in the Store 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'.

23.4   Store Web Monitor Source Page

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

```
2504558780: Source [0 10.29.3.42.14392 3958260924.1161732811]

    Topic: "test1"
    Session ID: 39307788
    Last Activity: 09:07:56.510005
    Repository: disk
        • Receiver Paced Persistence: 0
        • Message Map: 3120
        • Window: [0, 9d5, c2f]
        • Memory: 55986 / 65000 / 50331648
        • Age Threshold: 0
        • Sync: [c2f, c2f, c2f]
        • In Progress: 0 / 0
        • Offsets: 0 / 190320 / 4294967296
        • Active ULBs: 0 high 0
        • Loss: 0 ULBs 0
        • Drops: 0 / 0
    LBM Stats: [LBTRM:10.29.3.42:14390:12e46c8c:239.212.1.45:14400], received 609/35182, dups 0, loss 0, naks 0/0, ncfs
    0-0-0-0, unrec 0/0
    Receivers: 2504558781(39307788)
```

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 LBM 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"** or **"disk"**.
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.
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.

Source Page Item	Description
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 configuration file. • repository size limit, 50331648, is the Store's repository-size-limit topic option found in the Store 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 configuration file.
Sync: [c2f, c2f, c2f]	Pertains to disk 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.

Source Page Item	Description
In progress: 0 / 0	Pertains to disk 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.
Offsets: 0 / 190320 / 4294967296	Pertains to disk 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. • offset, 190320, The relative location of where the message, contig_sqn plus one will be written. • max_offset, 4294967296, The maximum size of the cache file.
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.

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

23.5   Store Web Monitor Receiver Page

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

```
2504558781: Receiver [0 10.29.3.42.14393 1510613393.1161732811]

     Topic: "test1"
     Last Activity: 09:09:35.981110
     Source RegID: 2504558780
     Source Session ID: 39307788
     ACK: c93
```

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 LBM 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.