Informatica™

Ultra Messaging (Version 6.16.1)

Concepts Guide

Contents

Chapter 1

Introduction

This document introduces the basic concepts and design approaches used by Ultra Messaging (UM).

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

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing at Informatica LLC 2100 Seaport Blvd. Redwood City, CA 94063.

Informatica products are warranted according to the terms and conditions of the agreements under which they are provided.
INFORMATICA LLC PROVIDES THE INFORMATION IN THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY OR CONDITION OF NON-INFRINGEMENT.

Chapter 2

Fundamental Concepts

Ultra Messaging is a software layer, supplied in the form of a dynamic library (shared object), which provides applications with message delivery functionality that adds considerable value to the basic networking services contained in the host operating system. The UMP and UMQ products also include a "Store" daemon that implements Persistence. The UMQ product also includes a "broker" daemon that implements Brokered Queuing.

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

Ultra Messaging is supported on a variety of platforms. There are three general categories of supported platforms:

- Core platforms: Linux, Windows, Solaris. These are built and shipped every release of UM.
- On-demand Platforms: AIX, HP-UX, Stratus (VOS), HP-NonStop (OSS), OpenVMS. These are only built and shipped by request from users.
- VM and Containers: UM can run in virtualized environments, including containers and VMs. See **Virtualized Environments**

In addition, Darwin (Mac OS) is supported (on demand) for development and test purposes. We do not support Darwin for production.

Note

> There are many different distributions of Linux and many different releases of all operating systems. Informatica does not certify UM on specific distributions or versions of operating systems. We do keep our test lab current and test on recent versions of the Core platforms. And we support users on the OS versions that they use in production. But we do not formally certify.

Applications access Ultra Messaging features through the Ultra Messaging Application Programming Interface (A↩PI). Ultra Messaging includes the following APIs: the UM C API, the UM Java API, and the UM .NET API. For details on these APIs, see:

- **UM C API**
- **UM Java API**
- `UM .NET API`

These APIs are very similar, and for the most part, this document concentrates on the C API. The translation from C functions to Java or .NET methods should be reasonably straightforward; see `Quick Start Guide` for sample applications in C, Java, and .NET. See also `C Example Source Code`, `Java Example Source Code`, and `C# Example Source Code`.

The UMQ product also supports the JMS API via the ActiveMQ broker.

The UM product is highly configurable to allow optimization over a wide variety of use cases. See Configuration Introduction for more information.

The three most important design goals of Ultra Messaging are to minimize message latency (the time that a given message spends "in transit"), maximize throughput, and ensure delivery of all messages under a wide variety of operational and failure scenarios. Ultra Messaging achieves these goals by not duplicating services provided by the underlying network whenever possible. Instead of implementing special messaging servers and daemons to receive and re-transmit messages, Ultra Messaging routes messages primarily with the network infrastructure at wire speed. Placing little or nothing in between the sender and receiver is an important and unique design principle of Ultra Messaging.

A UM application can function as a publisher (sometimes call a "source"), or a subscriber (sometimes called a "receiver"). A publishing application sends messages, and a subscribing application receives them. (It is also common for an application to function as both publisher and subscriber; we separate the concepts for organizational purposes.)

2.1 License Key

Before you can use UM, you must obtain a valid license key. A license key comes in two forms: a file and a string.

A license key file is a normal text file containing 4 lines. For example:

```
Product=UME,UMQ,LBM,UMDRO
Organization=My Company
Expiration-Date=never
License-Key=xxxx xxxx xxxx xxxx
```

A license key string is a single line of normal text which combines the above 4 lines, separated by colons. For example:

```
Product=UME,UMQ,LBM,UMDRO:Organization=My
    Company:Expiration-Date=never:License-Key=xxxx xxxx xxxx xxxx
```

The key (file or string) can be supplied to UM in two different ways: XML configuration file and environment.

2.1.1 License Via XML Configuration File

If you use **XML Configuration Files** to configure the UM library, you can specify a license key file using the **UM Element "<license>"** with format="filename". For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-configuration version="1.0">
  <license format="filename">my_um_license.txt</license>
...
```

Alternatively, you can supply the license information as a string in the XML with format="string":

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-configuration version="1.0">
  <license format="string">
    Product=UME,UMQ,LBM,UMDRO:Organization=My
        Company:Expiration-Date=never:License-Key=xxxx xxxx xxxx xxxx
  </license>
...
```

Note that you cannot supply a license key in **Plain Text Configuration Files** ("flat" files). See License Via Environment.

2.1.2 License Via Environment

You can specify a license key file using the environment variable LBM_LICENSE_FILENAME. For example:

```
export LBM_LICENSE_FILENAME="my_um_license.txt"
```

Alternatively, you can supply a license information as a string in the environment variable LBM_LICENSE_INFO. For example:

```
export LBM_LICENSE_INFO="Product=UME,UMQ,LBM,UMDRO:Organization=My
    Company:Expiration-Date=never:License-Key=xxxx xxxx xxxx xxxx"
```

2.2 Messaging Paradigms

UM supports three basic messaging paradigms (sometimes called Quality Of Service, or QOS):

- Streaming.
- Persistence.
- Queuing.

Sometimes people equate those paradigms with the UM product names "UMS", "UMP", and "UMQ". But this is not accurate. The UMP product supports both streaming and persistence. I.e. you can write streaming applications with UMP. The UMQ product supports streaming, persistence, and queuing. You can write streaming or persistent applications with UMQ.

2.2.1 Streaming

The UMS, UMP, and UMQ products support "Streaming" as their basic messaging paradigm. With Streaming, messages are sent directly from publisher to subscriber (no broker). A subscriber to a given topic joins the data stream of a publisher of that topic to receive messages. Messages sent during times that the subscriber is not joined are generally not available to the subscriber (live-only), although see Late Join.

2.2.2 Persistence

The UMP and UMQ products support Streaming and Persistence messaging paradigms. Persistence, sometimes called "durable" or "guaranteed" messages, saves messages sent by a publisher in non-volatile storage so that subscribers can recover missed messages under a variety of failure scenarios. If multiple subscribers exist for the same topic, each subscriber will get all messages sent by the publisher.

Persistence includes a component known as the persistent Store, which provides stable storage (disk or memory) of message streams.

UM delivers persisted messages from publisher to subscriber with very low latency by using the same technology as Streaming. This offers the functionality of durable subscriptions and confirmed message delivery.

(FYI - you will see references to "UME" in the configuration and APIs for persistence. "UME" is an earlier abbreviation for "UMP".)

For full details on UM Persistence, see the UM Guide for Persistence.

2.2.3 Queuing

The UMQ product supports Streaming, Persistence, and Queuing messaging paradigms. Queuing supports "load balancing" whereby published messages can be distributed across a set of subscribers such each message is only handled by one of the subscribers.

UMQ supports both Brokered queuing (where the Queue is a separate component which can store messages independent of the source and receiver), and Ultra Load Balancing (ULB), where the Queue is memory-based and resides in the source.

For full details on UM Queuing, see the `UM Guide to Queuing`.

2.3 Messages

The primary function of UM is to transport application messages from a publisher to one or more subscribers. For the purposes of UM, a message is a sequence of bytes, the parsing and interpretation of which is the responsibility of the application.

Since UM does not parse or interpret the messages, UM cannot reformat messages for architectural differences in a heterogeneous environment. For example, UM does not do byte-swapping between big and little endian CPUs.

However, there are some specific exceptions to this rule:

- The Self Describing Messaging feature is a separate library that allows the construction and parsing of structured messages as arbitrary name/value pairs.
- The Pre-Defined Messages feature is another separate library that allows the construction and parsing of structured messages.
- The Message Properties feature allows the setting of structured metadata which is sent along with an application message.

2.3.1 Message Integrity

To minimize latency and CPU overhead, UM relies on the Operating System and the Network equipment to ensure the integrity of message data. That is, UM does not add a checksum or digital signature to detect message corruption. Modern network hardware implements very reliable CRCs.

However, we have encountered a case of a microwave-based link that apparently did not use a strong error detection method. This user reported occasional packet corruptions.

We also encountered a situation where an Operating System and a network interface card were configured in an unusual way which led to reproducible undetected packet corruption.

Users who wish to reduce the possibility of message corruption to near-zero will want to implement some sort of message checksum or digital signature outside of UM. (Message Properties could be used to carry the checksum or signature.)

2.3.2 Message Metadata

It is sometimes useful for applications to send a message with additional metadata associated with the message. This metadata could be simply added to the message itself, but it is sometimes preferred that the metadata be separated from the message data.

For example, some organizations wrap UM in their own messaging middleware layer that provides a rich set of domain-specific functionality to their applications. This domain-specific functionality allows applications to be written more easily and quickly. However, that layer may need to add its own state-maintenance information which is independent from the application's message data. This middleware data can be supplied either as metadata attached to the message, or as separate non-application messages which are tagged with metadata. (The latter approach is often preferred when application messages must be sent with the minimum possible latency and overhead; adding metadata to a message adds processing overhead, so sending application messages without metadata is the most efficient.)

There are two different UM features that allow adding metadata to messages:

- Message Properties - a general system of typed name/value pairs.
- Application Headers - an older (deprecated) system of attaching small binary values.

Informatica generally recommends the use of Message Properties over Application headers.

2.4 Topic Structure and Management

UM offers the Publish/Subscribe model for messaging ("Pub/Sub"), whereby one or more receiver programs express interest in a topic ("subscribe"), and one or more source programs send to that topic ("publish"). So, a topic can be thought of as a data stream that can have multiple producers and multiple consumers. One of the functions of the messaging layer is to make sure that all messages sent to a given topic are distributed to all receivers listening to that topic. So another way of thinking of a topic is as a generalized destination identifier - a message is sent "to" a topic, and all subscribers receive it. UM accomplishes this through an automatic process known as topic resolution.

(There is an exception to this Publish/Subscribe model; see Immediate Messaging.)

A topic is just an arbitrary string. For example:

```
Orders
Market/US/DJIA/Sym1
```

It is not unusual for an application system to have many thousands of topics, perhaps even more than a million, with each one carrying a very specific range of information (e.g. quotes for a single stock symbol).

It is also possible to configure receiving programs to match multiple topics using wildcards. UM uses powerful regular expression pattern matching to allow applications to match topics in a very flexible way. Messages cannot be *sent* to wildcarded topic names. See UM Wildcard Receivers.

2.4.1 Message Ordering

UM normally ensures that received messages are delivered to the application in the same order as they were sent. However, this only applies to a specific topic from a single publisher. UM does not guarantee to retain order across different topics, even if those topics are carried on the same Transport Session. It also does not guarantee order within the same topic across different publishers. For users that need to retain order between different topics from a single publisher, see Spectrum.

Alternatively, it is possible to enforce cross-topic ordering in a very restrictive use case:

- The topics are from a single publisher (context),
- The topics are mapped to the same transport session,
- The Transport Session is configured for TCP (receiver-paced), IPC (receiver-paced), or SMX,
- The subscriber is in the same Topic Resolution Domain (TRD) as the publisher (no DRO in the data path),
- The messages being received are "live" - i.e. not being recovered from Late Join, OTR, or Persistence,
- The subscriber is not participating in Queuing,
- The subscriber is not using Hot Failover (HF).

2.4.2 Topic Resolution Overview

Topic Resolution ("TR") is a set of protocols and algorithms used internally by Ultra Messaging to establish and maintain shared state. Here are the basic functions of TR:

- Receiver discovery of sources.
- DRO routing information distribution.
- Persistent Store name resolution.
- Fault tolerance.

For more information, see Topic Resolution Description.

2.4.3 Topic Resolution Domain

A "Topic Resolution Domain" (TRD) is a set of applications and UM components which share the same Topic Resolution configuration and therefore participate in the TR protocols with each other. The key characteristic of a TRD is that all UM instances communicate directly with each other.

In small deployments of UM, a single TRD is all that is needed.

For larger deployments, especially deployments that are geographically separated with bandwidth-limited WAN links, the deployment is usually divided into multiple TRDs. Each TRD uses a different TR configuration, such that the applications in one TRD don't communicate directly applications in another TRD. The DRO is used to interconnect TRDs and provide connectivity between TRDs.

For more information, see Topic Resolution Description.

2.5 Messaging Reliability

Users of a messaging system expect every sent message to be successfully received and processed by the appropriately subscribed receivers with the lowest possible latency, 100% of the time. However, this would require perfect networks and computers that can handle unlimited load at infinite speed with complete reliability. Real world networks, computers, and software systems have limitations and are subject to overload and failure, which can lead to message loss.

One job of a messaging system is to detect lost messages and take additional steps to arrange their successful recovery. But again, the limits of hardware and software robustness can make 100% reliability impractical. Part of UM's power is to give the user tools to make intelligent trade-offs between reliability and other important factors, like memory consumption, delivery latencies, hardware redundancies, etc.

2.5.1 Unrecoverable Loss

There are two important concepts when talking about loss:

- Simple Loss - usually a case of lost packets due to an overloaded component. UM can be configured in a variety of ways to recover simple loss.
- Unrecoverable Loss - usually a case where a user-configured constraint has been exceeded and the messaging system has to give up trying to recover the lost data.

Simple loss is undesirable, even if the messaging system is able to recover the loss. Any recovery mechanism adds delay to the ultimate delivery of the message, and most uses have limits on the amount of time they are willing to wait for recovery. For example, consider a network hardware outage that takes 10 minutes to repair. A user might have a 5 minutes limit on the age of a message. Thus, the user would like messages sent during the first 5 minutes of the 10-minute outage to simply be declare unrecoverable. Messages sent during the last 5 minutes should be recovered and delivered.

However, when the messaging layer gives up trying to recover messages, it is important for the application software to be informed of this fact. UM delivers unrecoverable loss *events* to the application followed by subsequent messages successfully received. Applications can use those unrecoverable loss events to take corrective action, like informing the end user, and possibly initiating a re-synchronization operation between distributed components.

Obviously unrecoverable loss is considered a serious event. However, even simple loss should be monitored by users. Daily data rates tend to increase over time. A "clean" network this month might have overload-related simple loss next month, which might progress to unrecoverable loss the month after that.

UM does not deliver specific events to the application when simple loss is detected and successfully recovered. Instead, users have a variety of tools at their disposal to monitor UM transport statistics, which includes counters for simple loss. Applications themselves can use the UM API to "self-monitor" and alert end users when simple loss happens. Or external monitoring applications can be written to receive transport statistics from many applications, and provide a central point where small problems can be detected and dealt with before they become big problems.

See Packet Loss for an extended discussion of packet loss and how UM deals with it.

There are some special cases of unrecoverable loss that deserve additional description:

- Head Loss
- Leading Loss
- Tail Loss
- Burst Loss

2.5.2 Head Loss

When an application wants to publish messages, it creates one or more *UM sources* for topics. The design of UM is such that a subscriber *discovers* the sources of interest and *joins* them. The process of receivers discovering and joining sources (in UM it is called "Topic Resolution") takes a non-zero amount of time. Since UM is a fully-distributed system with no central master broker, the publisher has no way of knowing when the subscribers have completed the discovery/join process. As a result, it is possible for a publisher to create its sources and send messages, and some of those messages might not reach all of the subscribed receivers.

For many UM-based applications, this is not a problem. Consider a market data distribution system. The market data is a continuous stream of updates with no beginning or end. When a receiver joins, it has no expectation to get the "first" message; it joins at an arbitrary point in the stream and starts getting messages from that point.

But there are other applications where a sender comes up and wants to send a request to a pre-existing receiver. In this case, the sender is very interested in avoiding head loss so that the receiver will get the request.

UM's Persistence feature does a rigorous job of recovering from head loss. UM's Late Join feature is sometimes used by streaming receivers to recover from head loss, although it requires care in the case of a receiver restart since late join can also delivery duplicate messages.

2.5.3 Leading Loss

The behavior described in this section does not apply to Persisted data streams where delivery of all messages is important.

For Streaming (non-persisted) data streams, once a receiver successfully joins a source, it should start getting messages. However, there are some circumstances which interfere with the initial message reception.

For example, if a sender is sending a series of very large messages, those messages are fragmented, broken into smaller pieces and sent serially. If a receiver joins the message stream after the first message of a large message has already gone by, the receiver will no longer be able to successfully reassemble that first message. In UM versions prior to 6.12, this led to delivery of one or more Unrecoverable Loss events to the application receiver callback prior to delivery of the first successfully-received message.

Some users attempt to use the Late Join feature to avoid this problem, but the Late Join feature depends on a circular buffer of message fragments. Requesting Late Join may well recover the initial fragment of the message currently under transmission, but it might also recover the final fragments of the message before that. That leads to an unrecoverable loss event for that previous message.

As a final example, suppose that a receiver joins a source during a period of severe overload leading to packet loss. The receiver may not be able to get a full message until the overload condition subsides. This can deliver one or more unrecoverable loss events prior to the first successfully delivered message.

In all of these examples, UM has either gotten no messages or incomplete messages during the startup phase of a new receiver. Starting in UM version 6.12, UM will not deliver unrecoverable loss events to the application in these cases. Once UM is able to successfully deliver its first message to a receiver, UM will enable the delivery of unrecoverable loss events.

This matches what most programmers want to see. They are not interested in messages that came before their first, but they are interested in any gaps that happen after that first message.

Be aware that pre-6.12 versions of UM *do* deliver leading unrecoverable loss events to the application under some circumstances, leading application developers to implement their own filter for those leading events. Those application can safely upgrade to 6.12 and beyond; their filter will simply never be executed since UM filters the leading events first.

Finally, remember that it is important for UM-based systems to be monitored for transport statistics, including loss. Since leading unrecoverable loss events are now suppressed (and even pre-6.12 were not reliably delivered in all cases), the transport stats should be used to determine the "health" of the network.

2.5.4 Tail Loss

Packet loss is a fact of life. Temporary overloads of network hardware can lead to dropped packets. UM receivers are designed to recover those lost packets from the sender, but that recovery takes greater than zero time.

Suppose a application has some messages to send, and then exits. You could have a situation where the last message sent fell victim to a network drop. If the publisher exits immediately, the receivers may not have had enough time to recover the lost packets; may not even have detected the loss.

If the delivery of those tail messages is important, UM's persistence functionality should be used. A sender should delay its exit until the persistence layer informs it that all messages are "stable".

Non-persistent applications can implement an application-level handshake where the receivers tell the senders that they have successfully processed the final message.

Sometimes, delivery of the final message is not of critical importance. In that case, some application developers choose to simply introduce a delay of a second or two after the last message is sent before the sources are deleted. This will give UM a chance to detect and recover any lost messages.

See **Preventing NAK Storms with NAK Intervals** and **Preventing Undetected Unrecoverable Loss** for configuration details related to tail loss.

2.5.5 Burst Loss

UM's "burst loss" feature is no longer recommended. All known use cases for the burst loss feature are better accomplished through other means. For example, controlling latency after loss should be done using NAK-related and OTR-related configuration options. Informatica recommends disabling the burst loss feature by setting the configuration options **delivery_control_maximum_burst_loss (receiver)** and **delivery_control_maximum_burst_loss (hfx)** to 1,000,000,000.

When setting the burst loss options, be sure to include all subscribers, including all instances of UM Stores, DROs, and UMDS servers.

This is especially important for persistent receivers and Stores.

Feature Description

Normally, when the Delivery Controller detects a gap in topic sequence numbers of received message fragments, it waits for a NAK generation interval (defaults to 10 seconds) before declaring the missing message fragments unrecoverably lost. This wait time allows the underlying transport layer to attempt to retrieve the missing message fragments.

The configuration options **delivery_control_maximum_burst_loss (receiver)** and **delivery_control_maximum↩ _burst_loss (hfx)** specify a size for a contiguous gap in topic sequence numbers beyond which the gap is defined to be a "burst loss". When this happens, the delivery controller *immediately* declares the entire gap to be unrecoverably lost and resets its loss-handling structures.

Note that the burst loss feature does not reduce NAKs and retransmissions. If burst loss is invoked, the underlying transport layer will still attempt to recover the lost messages. NAK-based protocols will continue to send NAKs and retransmissions. However, the delivery controller will discard any successfully recovered messages because they were already declared unrecoverable as part of the burst loss event.

For burst loss, a single **LBM_MSG_UNRECOVERABLE_LOSS_BURST** event is delivered for the entire sequence number gap. Contrast this with simple (not burst) loss events, where a separate **LBM_MSG_UNRECOVERABL↩ E_LOSS** event will be delivered to the receiver for each lost sequence number.

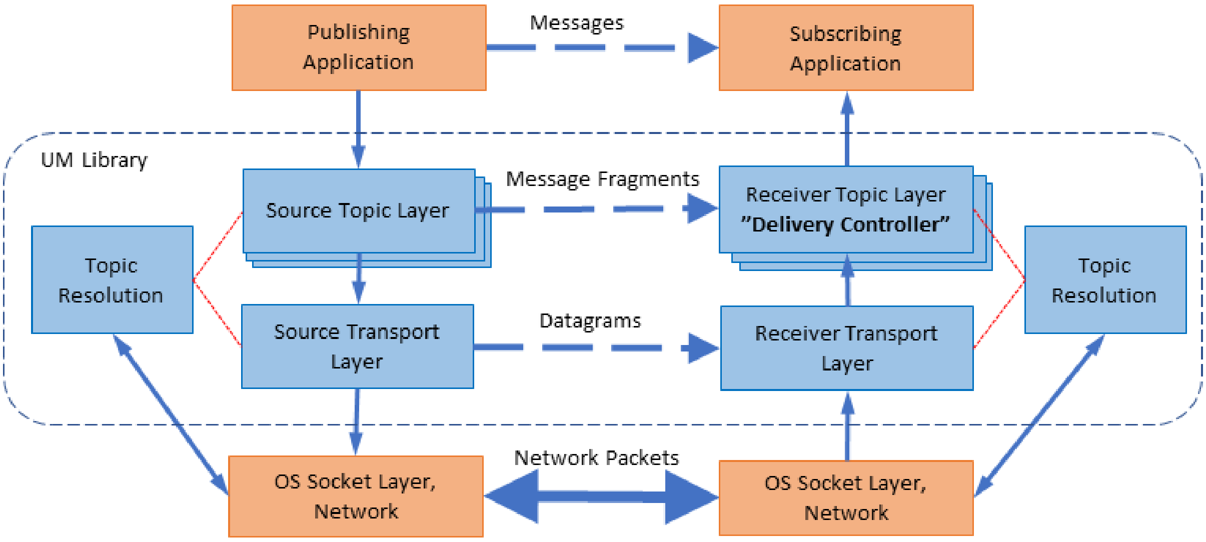
Note

> The burst loss control takes priority over all recovery methods. For example, if the subscriber is receiving persistent data and OTR is enabled, a gap longer than delivery_control_maximum_burst_loss will immediately declare the gap as unrecoverable without trying to use OTR to recover.

Finally, be aware that successfully-received but buffered messages can be discarded when a burst loss is detected. For example, let's say that topic sequence number 10 is lost, and number 11 through 30 are successfully received. Datagrams 11 through 30 will be buffered, awaiting datagram 10 being retransmitted. But if the next topic sequence number is 1055, that exceeds the burst loss threshold, which clears the entire order map, discarding messages 11 through 30. The application sees message 9, followed by a burst loss notification, followed by message 1055.

2.6 UM Software Stack

Here is a simplified diagram of the software stack:



At the bottom is the operating system socket layer, the computer hardware, and the network infrastructure. UM opens normal network sockets using standard operating system APIs. It expects the socket communications to operate properly within the constraints of the operational environment. For example, UM expects sent datagrams to be successfully delivered to their destinations, except when overload conditions exist, in which case packet loss is expected.

The UM library implements a Transport Layer on top of Sockets. The primary responsibility of the Transport Layer is to reliably route datagrams from a publishing instance instance of UM to a receiving instance of UM. If datagram delivery fails, the Transport Layer detects a gap in the data stream and arranges retransmission.

UM implements a Topic Layer on top of its Transport Layer. Publishers usually map multiple topics to a Transport Session, therefore there can be multiple instances of the topic layer on top of a given transport layer instance ("↩ Transport Session"). The Topic layer is responsible for UM fragmentation of messages (splitting large application messages into datagram-sized pieces) and sending them on the proper Transport Session. On the receiving side, the Topic Layer (by default) reassembles fragmented messages, makes sure they are in the right order, and delivers them to the application. Note that the receive-side Topic Layer has a special name: the *Delivery Controller*.

In addition to those layers is a Topic Resolution module which is responsible for topic discovery and triggering the receive-side joining of Transport Sessions.

2.6.1 Delivery Controller

The interaction between the receiver Transport Layer and the Delivery Controller (receive-side Topic Layer) deserves some special explanation.

In UM, publishing applications typically map multiple topic sources to a Transport Session. These topics are multiplexed onto a single Transport Session. A subscribing application will instantiate an independent Delivery Controller for each topic source on the Transport Session that is subscribed. The distribution of datagrams from the Transport Session to the appropriate Delivery Controller instances is a de-multiplexing process.

In most communication stacks, the transport layer is responsible for both reliability and ordering - ensuring that messages are delivered in the same order that they were sent. The UM division of functionality is different. It is the Delivery Controller which re-orders the datagrams into the order originally sent.

The transport layer delivers datagrams to the Delivery Controller in the order that they arrive. If there is datagram loss, the Delivery Controller sees a gap in the series of topic messages. It buffers the post-gap messages in a structure called the *Order Map* until transport layer arranges retransmission of the lost datagrams and gives them to the Delivery Controller. The Delivery Controller will then deliver to the application the re-transmitted message, followed by the buffered messages in proper order.

To prevent unbounded memory growth during sustained loss, there are two configuration options that control the size and behavior of the Order Map: **delivery_control_maximum_total_map_entries (context)** and **otr_message_caching_threshold (receiver)**.

This is an important feature because if a datagram is lost and requires retransmission, significant latency is introduced. However, because the Transport Layer delivers datagrams as they arrive, the Delivery Controller is able to deliver messages for topics that are unaffected by the loss. See Example: Loss Recovery for an illustration of this.

This design also enables the UM "Arrival Order Delivery" feature directly to applications (see Ordered Delivery). There are some use cases where a subscribing application does not need to receive every message; it is only important that it get the *latest* message for a topic with the lowest possible latency. For example, an automated trading application needs the latest quote for a symbol, and doesn't care about older quotes. With Arrival Order delivery, the transport layer will attempt to recover a lost datagram, an unavoidable latency. While waiting for the retransmission, a newer datagram for that topic might be received. Rather than waiting for the retransmitted lost datagram, the Delivery Controller will immediately deliver the newer datagram to the application. Then, when the lost datagram is retransmitted, it will also be delivered to the application. (Note: with arrival order delivery, the Order Map does not need to buffer any messages since all messages are delivered immediately on reception.)

Applications can track when Delivery Controllers are created and deleted using the **source_notification_function (receiver)** configuration option. This is generally preferable to using Receiver BOS and EOS Events.

2.7 UM Threading

Ultra Messaging is designed for event-driven, asynchronous, distributed applications. Threads are used to implement the asynchronous behavior, with application callbacks being made from UM threads independently from application threads.

The UM Context Object owns a thread called the "context thread". This thread waits for network activity and handles much of the protocol work for UM. In particular, the context thread is responsible for reading messages from the network socket, preparing them, and delivering them to the application's receiver callback function.

However, the context thread often is not involved in the sending of application messages. An application thread calls the UM send function, and in a typical case, that thread flows through the UM code and makes the socket "send" call. An exception to this is if Implicit Batching is used and a timeout trigger the sending of a partial batch; in this case it is the context thread that sends the message.

The context thread can either be created implicitly when the context is created (Embedded Mode), or the application can choose to create the context thread external to UM and "donate" it to a context (Sequential Mode).

There are some Other Specialized Threads that the context can create, depending on which UM features you use.

2.7.1 Embedded Mode

When you create a context (**lbm_context_create()**) with the UM configuration option **operational_mode (context)** set to embedded (the default), UM creates an independent thread, called the context thread, which handles timer and socket events, and does protocol-level processing, like retransmission of dropped packets.

2.7.2 Sequential Mode

Ultra Messaging typically relies on certain independent threads to perform a variety of its functions. For example, network messages are received by either a context thread or an XSP thread.

By default, UM creates the necessary threads internally using the operating system's thread primitives. However, there are times where an application needs more control. Here are two example use cases:

1. The application wants to create the threads so that specific thread attributes can be set as part of creating the threads. For example, CPU affinity, execution priority, even stack size.

2. The application wants to serialize all UM operations without the use of any independent threads of execution. In this case, the application may want to have a single thread which executes application code and UM code serially.

These goals can be met using "sequential" mode. With this mode, UM does not create the thread. It becomes the application's responsibility to call the event processing API. For use case 1 above, the application typically creates its own independent thread which simply calls the event processing API in a loop. For use case 2 above, the application calls the event processing API from its main thread.

2.7.3 Context Sequential Mode

When you create a context (C API: **lbm_context_create()**), by default a separate independent thread is created. This is the "context thread".

You enable Sequential mode by setting the configuration option **operational_mode (context)** to the value "sequential". For example:

```
context operational_mode sequential
```

The event processing C API for the context is **lbm_context_process_events()** or **lbm_context_process_events↩_ex()**. Java: **LBMContext.processEvents()**.

Warning

> If using sequential mode and **fd_management_type** "wincompport", the thread that creates the context must not exit while the context is active. Furthermore, you gain a small performance improvement if the thread that creates the context is the same thread that calls **lbm_context_process_events()**.

2.7.4 XSP Sequential Mode

An XSP is a specialized form of a context. When you create an XSP (C API: **lbm_xsp_create()**), by default a separate independent thread is created. This is the "XSP thread".

You enable Sequential mode by setting the configuration option **operational_mode (xsp)** to the value "sequential". For example:

```
xsp operational_mode sequential
```

The event processing C API for XSP is **lbm_xsp_process_events()**. Java: **LBMXSP.processEvents()**.

Warning

> If using sequential mode and **fd_management_type** "wincompport", the thread that creates the XSP must not exit while the context is active. Furthermore, you gain a small performance improvement if the thread that creates the context is the same thread that calls **lbm_xsp_process_events()**.

2.7.5 IPC Sequential Mode

When a receiver joins an IPC source's transport, by default the context dynamically creates an independent thread to service the IPC shared memory. This is the "IPC thread".

You enable Sequential mode by setting the configuration option **transport_lbtipc_receiver_operational_mode (context)** to the value "sequential". For example:

```
context transport_lbtipc_receiver_operational_mode sequential
```

The event processing C API for IPC is **lbm_context_process_lbtipc_messages()**. The Java API does not support IPC sequential mode.

2.7.6 Other Specialized Threads

In addition to the context thread, there are other UM features which rely on specialized threads:

- The Transport LBT-SMX feature, when used, creates its own specialized receive thread. However, unlike the context thread and the LBT-IPC threads, the creation of the LBT-SMX thread is handled by UM. There is no sequential mode for the LBT-SMX thread.

- The **DBL transport acceleration** feature, when used, creates its own specialized receive thread. However, unlike the context thread and the LBT-IPC threads, the creation of the DBL thread is handled by UM; there is no sequential mode for the DBL thread.

- The **Automatic Monitoring** feature, when used, creates an independent context to publish statistics via UM, and a second internal thread for sampling data. The creation of the automatic monitoring threads is handled by UM; there is no sequential mode for the automatic monitoring threads.

- The TCP-based Topic Resolution, when used, creates its own thread to handle communication with the SR↩ Ses.

2.8 Deleting UM Objects

Before you can delete a given UM object, you must first delete all associated objects. For example, before deleting a context, you must delete all sources and receivers associated with that context.

The general order of object deletion is:

1. **Cancel UM Timers** ("cancel" instead of "delete").
2. **Delete any outstanding request objects**.
3. **Delete sources and receivers, and cancel Registered File Descriptors**. ("Receivers" includes wildcard, hf, and hfx receivers.)
4. **Delete context(s)**.
5. **Delete event queue(s)**. See "Deleting an Event Queue" for more information.

Some important warnings:

- When deleting a source object, any messages waiting to be sent due to implicit batching or rate limiter might not be sent (if deleting the source also deletes the transport session). Also, any subscribers sending retransmission requests will be ignored.
- For objects that have application callbacks (e.g. receivers), there can be a race condition where the object deletion function returns but an application callback is still executing. See the Callback After Delete.
- In UM's Java and .NET APIs, each object should be explicitly deleted, usually by calling its "close" method. You should not simply release all references to a UM object and expect the garbage collector to do the cleanup.

2.8.1 Callback After Delete

For objects with application callbacks, there are circumstances where the delete function successfully returns to the caller while its callback is still executing. This can lead to race conditions and hard-to-diagnose bugs. For example, an application might delete a receiver object and then immediately delete some resources (memory, etc) that the receiver callback uses. But if the receiver callback is executing at the same time that the resources are being deleted, you can get corrupted memory and crashes.

For most object types, this can only happen if the deleted object is associated with an event queue. However, for UM Timers and Registered File Descriptors, you can have callback execution continuing after the cancel returns even if no event queue is used.

One effective technique of avoiding this race condition is called "event sync". An alternative for C programs is to use the C API Extended Delete.

2.8.2 Event Sync

The "event sync" technique works with all APIs and object types, with or without an event queue. It is not a separate UM feature; rather it is a technique of using a UM timer to synchronize threads.

The event sync technique consists of deleting one or more objects, and then scheduling a zero-duration timer with a "deletion complete" callback. The timer is associated with the same context as the object(s) being deleted. If the object(s) being deleted are associated with an event queue, the timer should be associated with the same event queue.

Since the timer has a duration of zero, it expires immediately. The UM context thread and/or event queue will finish whatever it is doing and expire the timer. For the non-event queue case, the timer expiration callback is executed immediately by the context. If the timer is associated with an event queue, the dispatch thread will finish whatever it is doing and execute the timer expiration callback. This guarantees that any executing callback for the object being deleted will be completed before the event sync callback is executed.

It is common for a program shutting down to delete hundreds, or even thousands of source and/or receiver objects. It is usually not necessary to perform an event sync for each object. You can delete all of the source and receiver objects and then perform a single event sync to guarantee that all callbacks are done. Then resources used by the callbacks can be safely cleaned up.

See `event_sync` for C and Java examples.

2.8.3 C API Extended Delete

For a C application deleting an object that is associated with an event queue, there is an "extended" delete function which notifies the application via callback when UM guarantees no further callbacks are executing.

The C functions are:

- **lbm_context_delete_ex()**
- **lbm_cancel_timer_ex()**
- **lbm_cancel_fd_ex()**
- **lbm_src_delete_ex()**
- **lbm_rcv_delete_ex()**
- **lbm_request_delete_ex()**
- **lbm_wildcard_rcv_delete_ex()**
- **lbm_hf_rcv_delete_ex()**
- **lbm_hfx_delete_ex()**
- **lbm_hfx_rcv_delete_ex()**

Note that the "cancel callback" might be invoked synchronously with the call to the ..._ex() function, or it might be invoked asynchronously by the event queue dispatch thread.

Note that the Java and .NET APIs do not have this extended form of object deletion; Informatica recommends using Event Sync.

2.9 DRO

The Ultra Messaging Dynamic Routing Option (DRO) consists of a daemon named "tnwgd" that bridges disjoint Topic Resolution Domains (TRDs) by effectively forwarding control and user traffic between them. Thus, the DRO facilitates WAN routing where multicast routing capability is absent, possibly due to technical obstacles or enterprise policies.

The DRO transfers multicast and/or unicast topic resolution information, thus ensuring that receivers in disjoint topic resolution domains from the source can receive the topic messages to which they subscribe.

See the `Dynamic Routing Guide` for more information.

2.10 Late Join

In many applications, a new receiver may be interested in messages sent before that receiver joins the source's transport session. The Ultra Messaging Late Join feature allows a new receiver to obtain previously-sent messages from a source. Without the Late Join feature, the receiver would only deliver messages sent after the receiver successfully joins the source's transport session. With Late Join, the source locally stores recently sent messages according to its Late Join configuration options, and a new receiver is able to retrieve those messages.

For late join to happen, the source must be configured to offer it, and the receiver must be configured to participate.

Source-side configuration options:

- **late_join (source)**
- **retransmit_retention_age_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_size_threshold (source)**
- **request_tcp_interface (context)**

Receiver-side configuration options:

- **use_late_join (receiver)**
- **retransmit_request_interval (receiver)**
- **retransmit_request_message_timeout (receiver)**
- **retransmit_request_outstanding_maximum (receiver)**
- **late_join_info_request_interval (receiver)**
- **late_join_info_request_maximum (receiver)**
- **retransmit_initial_sequence_number_request (receiver)**
- **retransmit_message_caching_proximity (receiver)**
- **response_tcp_interface (context)**

Note

With Smart Sources, the following configuration options have limited or no support:

- **retransmit_retention_size_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_age_threshold (source)**

You cannot use Late Join with Queuing functionality (UMQ).

2.11 Request/Response

Ultra Messaging also offers a Request/Response messaging model. A publisher (the requester) sends a message to a topic. Every receiving application listening to that topic gets a copy of the request. One or more of those receiving applications (responder) can then send one or more responses back to the original requester. Ultra Messaging sends the request message via the normal pub/sub method, whereas Ultra Messaging delivers the response message directly to the requester.

An important aspect of the Ultra Messaging Request/Response model is that it allows the application to keep track of which request corresponds to a given response. Due to the asynchronous nature of Ultra Messaging requests, any number of requests can be outstanding, and as the responses come in, they can be matched to their corresponding requests.

Request/Response can be used in many ways and is often used during the initialization of Ultra Messaging receiver objects. When an application starts a receiver, it can issue a request on the topic the receiver is interested in. Source objects for the topic can respond and begin publishing data. This method prevents the Ultra Messaging source objects from publishing to a topic without subscribers.

Be careful not to be confused with the sending/receiving terminology. Any application can send a request, including one that creates and manages Ultra Messaging receiver objects. And any application can receive and respond to a request, including one that creates and manages Ultra Messaging source objects.

Note

> You cannot use Request/Response with Queuing functionality.

For more details, see Request/Response Model.

2.12 Registered File Descriptors

The UM C API running on Unix supports an application creating its own network socket and registering it with the UM context. This allows state changes of the file descriptor (e.g. readable, writable) to be monitored by the UM context. The application provides a callback function that UM invokes when the file descriptor state changes.

This is a rarely-used feature of UM, mostly by market data feed handlers.

Java and .NET sockets are not supported. Windows is also not supported.

To register and cancel an application file descriptor with a context, use:

- **lbm_register_fd()**
- **lbm_cancel_fd()** or **lbm_cancel_fd_ex()**

To register and cancel an application file descriptor with an XSP, use:

- **lbm_xsp_register_fd()**
- **lbm_xsp_cancel_fd()** (no "_ex()" form is available)

When canceling a registration, the guidelines for Deleting UM Objects must be followed. Pay special attention to Callback After Delete.

2.13 UM Transports

A source application uses a UM transport to send messages to a receiver application. An Ultra Messaging transport type is built on top of a standard IP protocol. For example, the UM transport type "LBT-RM" is built on top of the standard UDP protocol using standard multicast addressing. The different Ultra Messaging transport types have different trade-offs in terms of latency, scalability, throughput, bandwidth sharing, and flexibility. The publisher chooses the transport type that is most appropriate for the data being sent, at the topic level. A programmer might choose different transport types for different topics within the same application.

Se Transport Types for the different types of UM transports available.

2.14 Transport Sessions

An Ultra Messaging publisher can make use of very many topics - possibly over a million. Ultra Messaging maps those topics onto a much smaller number of *Transport Sessions*. A Transport Session can be thought of as a specific running instance of a transport type, running within a context. A given Transport Session might carry a single topic, or might carry tens of thousands of topics.

A publishing application can either explicitly map each topic source to specific Transport Sessions, or it can make use of an automatic mapping of sources to a default pool of Transport Sessions. If explicitly mapping, the application must configure a new source with identifying information to specify the desired Transport Session. The form of this identifying information depends on the transport type. For example, in the case of the LBT-RM transport type, a Transport Session is identified by a **multicast group IP address** and a **destination port number**. Alternatively, if the application does not specify a Transport Session for a new topic source, a Transport Session is implicitly selected from the default pool of Transport Sessions, configured when the context was created. For example, with the LBT-RM transport type, the default pool of implicit Transport Sessions is created with a range of multicast groups, from **low** to **high**, and the **destination port number**. Note that at context creation, the Transport Sessions in the configured default pool are not immediately activated. As topic sources are created and mapped to Transport Sessions, those Transport Sessions are activated.

There are many cases where a transport session is identified using a Source String (minus the bracketed topic index).

Note

> When two contexts are in use, each context may be used to create a topic source for the same topic name. These sources are considered separate and independent, since they are owned by separate contexts. This is true regardless of whether the contexts are within the same application process or are separate processes. A Transport Session is also owned by a context, and sources are mapped to Transport Sessions within the same context. So, for example, if application process A creates two contexts, ctx1 and ctx2, and creates a source for topic "current_price" in each context, the sources will be mapped to completely independent Transport Sessions. This can even be true if the same Transport Session identification information is supplied to both. For example, if the source for "current_price" is created in ctx1 with LBT-RM on multicast group 224.10.10.10 and destination port 14400, and the source for the same topic is created in ctx2, also on LBT-RM with the same multicast group and destination port, the two Transport Sessions will be separate and independent, although a subscribing application will receive both Transport Sessions on the same network socket.

2.14.1 Subscribing to a Transport Session

A receiving application might subscribe to a small subset of the topics that a publisher has mapped to a given Transport Session. In most cases, the subscribing process will receive all messages for all topics on that Transport Session, and the UM library will discard messages for topics not subscribed. This user-space filtering does consume system resources (primarily CPU and bandwidth), and can be minimized by carefully mapping topics onto Transport Sessions according to receiving application interest (having receivers). (Certain transport types allow that filtering to happen in the publishing application; see **transport_source_side_filtering_behavior (source)**.)

When a subscribing application creates its first receiver for a topic, UM will join any and all Transport Sessions that have that topic mapped. The application might then create additional receivers for other topics on that same Transport Session, but UM will not "join" the Transport Session multiple times. It simply sets UM internal state indicating the topic subscriptions. When the publisher sends its next message of any kind on that Transport Session, the subscribing UM will deliver a BOS event (Beginning Of Stream) to all topic receivers mapped to that Transport Session, and will consider the Transport Session to be *active*. Once active, any subsequent receivers created for topics mapped to that same Transport Session will deliver an immediate BOS to that topic receiver.

If the publisher deletes a topic source, the subscribing application may or may not get an immediate EOS event (End Of Stream), depending on different circumstances. For example, in many cases, the deletion of topic sources by a publisher will not trigger an EOS event until *all* sources mapped to a Transport Session are deleted. When the last topic is deleted, the Transport Session itself is deleted, and an EOS event might then be delivered to *all* topic receivers that were mapped to that Transport Session. (The use of **resolver_send_final_advertisements (source)**, TCP-Based Topic Resolution Details, or DROs can change this behavior.) Note that for UDP transports, the deletion of a Transport Session by the publisher is not immediately detected by a subscriber, but rather requires the expiration of an activity timeout.

Be aware that in a deployment that includes the DRO, BOS and EOS may only indicate the link between the receiver and the local DRO portal, not necessarily full end-to-end connectivity. Subscribing application should not use BOS and EOS events as an accurate and timely indication of the creation and deletion of sources by a publisher.

Note

> Non-multicast Ultra Messaging transport types can use source-side filtering to decrease user-space filtering on the receiving side by doing the filtering on the sending side. However, be aware that system resources consumed on the source side affect all receivers, and that the filtering for multiple receivers must be done serially, whereas letting the receivers do the filtering allows that filtering to be done in parallel, only affecting those receivers that need the filtering.

2.14.2 Transport Session Differences

There are a few subtle differences in behavior of transport sessions depending on which transport type is being used.

As sources are created but not explicitly assigned to a specific transport session, UM will assign the source to a transport session in the default pool, in a "round-robin" fashion. But how is the default pool populated with transport sessions?

- Transport LBT-RM - the range of multicast groups from **transport_lbtrm_multicast_address_low (context)** to **transport_lbtrm_multicast_address_high (context)** defines the default pool. Note that an LBT-RM transport session is also specified with a destination port, but that port is not specified as a range and is not iterated to create multiple transport sessions.

- Transport LBT-IPC and Transport LBT-SMX - the range of IDs, low to high, defines the default pool. See **transport_lbtipc_id_low (context)** to **transport_lbtipc_id_high (context)** for IPC, or **transport_lbtsmx↩_id_low (context)** to **transport_lbtsmx_id_high (context)** for SMX.

- Transport TCP, Transport LBT-RU - both transport types have a range of ports that the default pool is populated from, but the number of transport sessions in the default pool is not simply the port low-to-high difference. Instead, both have a "maximum ports" option that limits the number of transport sessions automatically created. This is because the port low-to-high range is often made large so that multiple application processes can share from it, with each process limited to the maximum ports setting. See **transport_tcp_port_low (context)** to **transport_tcp_port_high (context)** and **transport_tcp_maximum_ports (context)** for TCP, or **transport_lbtru_port_low (context)** to **transport_lbtru_port_high (context)** and **transport_lbtru_↩ maximum_ports (context)** for LBT-RU.

An application can configure a source to a specific transport session, either in the default pool, or outside of it. For example, an application might use **transport_lbtrm_multicast_address (source)** to assign a source to a multicast group that is outside of the low-to-high range.

The different source transport types differ in how this kind of "outside of the default pool" assignment affects the transport session default pool:

- Transport LBT-RM, Transport LBT-IPC, and Transport LBT-SMX - the explicitly added transport session outside of the configured default pool range does not affect the default pool. For example, assigning a source to a multicast group outside of the low-to-high range does not affect the default pool. Subsequent sources created without a specific multicast group will only be assigned to groups within the low-to-high range. Assigning a source to a transport session that is outside the low-to-high range will prevent other sources from sharing that transport session.

- Transport TCP, Transport LBT-RU - the explicitly added transport session will be added to the default pool. For example, if the low port is 12000 and the high port is 12049 and the maximum ports if 5, the publishing context might assign ports 12000 - 12004 to the default pool. If the publisher then creates a source assigned to port 12020 and another assigned to port 12099, both of those transport sessions will be added to the default pool, making it 7 transport sessions. In particular, creating a source without a specific port assigned will be automatically assigned to one of the default pool transport sessions, and so might be assigned to port 12020 or 12099. Assigning a source to a transport session outside of the low-to-high range does not prevent other sources from sharing the transport session.

See the configuration section for each transport type for specifics on how Transport Sessions are created:

- **TCP Transport Session Management**
- **LBT-RM Transport Session Management**
- **LBT-RU Transport Session Management**
- **LBT-IPC Transport Session Management**
- **LBT-SMX Transport Session Management**

With the UMQ product, a ULB source makes use of the same transport types as Streaming, but a Brokered Queuing source must use the **broker** transport.

2.15 Transport Pacing

Pacing refers to the controlling of when messages can be sent. *Source Pacing* is when the publisher of messages controls when messages are sent, and *Receiver Pacing* is when the subscribers can essentially push back on the publisher if the publisher is sending faster than the subscribers can consume messages.

2.15.1 Source Pacing

In its simplest case, source pacing means that the publishing application controls when to send messages by having messages to send. For example, market data distribution system sends messages when the exchanges it is monitoring sends updates. The subscribers must be able to accept those messages at the rates that the exchanges send them.

A slow subscriber will fall behind and can lose data if buffers overflow. There is no way for a slow subscriber to "push back" on a fast publisher.

Some of Ultra Messaging's transport types have features that allow the publisher to establish upper limits on the outgoing message data. For example, LBT-RM has the **transport_lbtrm_data_rate_limit (context)** configuration option. If the application attempts to send too many messages in too short a time, the send function can block for a short time, or return an error indicating that it attempted to exceed the configured rate limit. This is still called source pacing because the rate limit is optional and is under the control of the publisher.

The source-paced transports are:

- Transport TCP (can be **configured** for source or receiver pacing)
- Transport LBT-RU (source pacing only)
- Transport LBT-RM (source pacing only)
- Transport LBT-IPC (can be **configured** for source or receiver pacing)

2.15.2 Receiver Pacing

In a receiver-paced transport, a publisher still attempts to send when it has messages to send, but the send function can block or return an error if the publisher is sending faster than the slowest subscribed receiver can consume. There is a configurable amount of buffering that can be used to allow the publisher to exceed the consumption rate for short periods of time, but when those buffers fill, the sender will be prevented from sending until the buffers have enough room.

The receiver-paced transports are:

- Transport TCP (can be **configured** for source or receiver pacing)
- Transport LBT-IPC (can be **configured** for source or receiver pacing)
- Transport LBT-SMX (receiver-pacing only)

However, it is generally inadvisable to rely on receiver pacing to ensure reliable system operation. Subscribers should be designed and provisioned to support the maximum possible incoming data rates. The reasons for this recommendation are described below.

2.15.3 Receiver Queuing

This use case is not related to the Queuing paradigm. It is referring to the use of a software queue between the context thread which receives the UM message and the application thread which processes the message.

A receiver-paced transport does not necessarily mean that a slow application will push back on a fast publisher. Suppose that the actual message reception action by UM's context thread is fast, and the received messages are placed in an unbounded memory queue. Then, one or more application threads take messages from the queue to process them. If those application threads don't process messages quickly enough, instead of pushing back on the publisher, the message queue will grow.

For example, the UM Event Queue Object is an unbounded memory-based queue. You could use Transport TCP configured for receiver pacing, but if the subscriber uses an event queue for message delivery, then a slow consumer will not slow down the publisher. Instead the subscriber's event queue will grow without bound, eventually leading to an out-of-memory condition.

2.15.4 Pacing and DRO

Transport pacing refers to the connection within a a Topic Resolution Domain (TRD) from source to receiver. If multiple TRDs are connected with a DRO, there can be multiple hops between the originating publisher and the final subscriber, with the DROs providing proxy sources and receivers to pass messages across the hops. The pacing of a particular transport session only applies to the link between the source (or proxy source) and the receiver (or proxy receiver) of a particular hop.

This means that UM does not support end-to-end transport pacing.

For example, if a DRO joins two TRDs, both of which use receiver-paced TCP for transport, a slow receiver might push back on the DRO's proxy source, leading to queuing in the DRO. Since the DRO uses bounded queues, the DRO's queue will fill, leading to dropped messages.

2.15.5 Pacing and Queuing

Brokered Queuing is receiver paced between the source and the broker. For ULB queuing, the transport pacing is determined by the chosen transport type.

There is also a form of end-to-end application receiver-pacing with queuing. For both brokered ULB, the size of queue is configurable. As receivers consume messages, they are removed from the queue. If the queue fills due to the source exceeding the consumption rate of the receiver, the next send call will either block or return a "WOULDBLOCK" error until sufficient room opens up in the queue. Note that lifetimes can be configured for messages (to prevent unbounded blocking) and the application can be notified if a message exceeds its lifetime.

See `UM Guide to Queuing` for more information.

2.15.6 Pacing and Persistence

With Persistence, the pacing is layered:

- At the transport level, pacing is determined by the chosen transport type.
- At the persistence level, Transport Pacing provides receiver pacing between the source and the Store. This helps to ensure that a publisher sending on a source-paced transport like LBT-RM will not overwhelm the Store.
- By default, the end-to-end pacing is source-paced. Which is to say that if a subscribed receiver cannot keep up with the publish rate, eventually the receiver can experience unrecoverable loss, although this normally will only happen if the receiver falls further behind than the total size of the Store's repository.
- Alternatively, the end-to-end pacing can be switched to receiver-paced. This is done using **RPP: Receiver-Paced Persistence**. Note that if a subscribed receiver crashes, it can lead to unbounded blockage of the publisher until the subscriber is successfully restarted.

2.15.7 Suspended Receiver Problem

This is a problem that one of our users encountered. Their system used receiver-paced TCP because they wanted the publishing speed to be limited to the rate that the slowest receiver could consume. This was fine until one day the publisher appeared to stop working.

With some help from Support, the problem was trace to a receiver at a particular IP address that was joined to the transport but apparently was not reading any data. It took some time to find the receiver by its IP address, and what they found was a Windows machine with a dialog box telling the user that the process had generated an exception and did the user want abort the process or enter the debugger.

When Windows does this, it suspends the process waiting for user input. In that suspended state, the kernel maintains the network connections, but the process is not able to read any data. So the socket buffers filled up and the publisher's send call blocked.

2.16 Event Delivery

There are many different events that UM may want to deliver to the application. Many events carry data with them (e.g. received messages); some do not (e.g. end-of-stream events). Some examples of UM events:

- Received message. UM delivers messages on subscribed topics to the receiver.
- Receiver loss. UM can inform the application when a data gap is detected on a subscribed topic that could not be recovered through the normal retransmission mechanism.
- End of Stream. UM can inform a receiving application when a data stream (Transport Session) has terminated.
- Connect and Disconnect events. Source-side events delivered to a publisher when receivers connect to and disconnect from the source's transport session.
- A timer expiring. Applications can schedule timers to expire in a desired number of milliseconds (although the OS may not deliver them with millisecond precision).
- An application-managed file descriptor event. The application can register its own file descriptors with UM to be monitored for state changes (readable, writable, error, etc.).
- New source notification. UM can inform the application when sources are discovered by Topic Resolution.

UM delivers events to the application via callbacks. The application explicitly gives UM a pointer to one of its functions to be the handler for a particular event, and UM calls that function to deliver the event, passing it the parameters that the application requires to process the event. In particular, the last parameter of each callback type is a client data pointer (clientdp). This pointer can be used at the application's discretion for any purpose. It's value is specified by the application when the callback function is identified to UM (typically when UM objects are created), and that same value is passed back to the application when the callback function is called.

There are two methods that UM can use to call the application callbacks: through context thread callback, or event queue dispatch.

In the context thread callback method (sometimes called direct callback), the UM context thread calls the application function directly. This offers the lowest latency, but imposes significant restrictions on the application function. See Event Queue Object.

The event queue dispatch of application callback introduces a dynamic buffer into which the UM context thread writes events. The application then uses a thread of its own to dispatch the buffered events. Thus, the application callback functions are called from the application thread, not directly from the context thread.

With event queue dispatching, the use of the application thread to make the callback allows the application function to make full, unrestricted use of the UM API. It also allows parallel execution of UM processing and application processing, which can significantly improve throughput on multi-processor hardware. The dynamic buffering provides resilience between the rate of event generation and the rate of event consumption (e.g. message arrival rate v.s. message processing rate).

In addition, an UM event queue allows the application to be warned when the queue exceeds a threshold of event count or event latency. This allows the application to take corrective action if it is running too slow, such as throwing away all events older than a threshold, or all events that are below a given priority.

Note that while most UM events can be delivered via event queue, there are some that cannot (e.g. **resolver_source_notification_function (context)**).

2.16.1 Receiver BOS and EOS Events

There are two receive-side events that some applications use but which are not recommended for most use cases:

- **LBM_MSG_BOS** - Beginning of (transport) Session.
- **LBM_MSG_EOS** - End of (transport) Session.

The BOS and EOS events are delivered to receivers on a topic basis, but were originally designed to represent the joining and exiting of the underlying Transport Session.

Note that the BOS and EOS receive-side events are very similar to the Source Connect and Disconnect Events.

The BOS event means slightly different things for different transport types. The basic principal is that a UM receiver won't deliver BOS until it has seen some kind of activity on the transport after joining it. For example, a session message for Transport LBT-RM. Or perhaps a TSNI on any of the transport types. Or even an actual user message. Any of these will trigger BOS. But if the newly-joined transport session is completely silent, then BOS is delayed until first activity is detected. Under some configurations, that delay is unbounded.

EOS is also triggered by different things for different transport types. For example, for the TCP transport, a disconnect will trigger EOS for any receivers that were subscribed to topics carried on that transport session. For LBT-RM, an **activity timeout** can trigger EOS.

However, in recent versions of UM, the meaning of EOS has become more complicated, and no longer necessarily indicates that the transport session has ended. There are circumstances in many versions of UM where a publisher of a source that is in a remote TRD can delete that source, and the receiver will receive a timely EOS, even though the transport session that carried the source stays in effect and joined by the receiver. As of UM version 6.10, the configuration option **resolver_send_final_advertisements (source)** can be used to trigger EOS on receivers in the same TRD, even if the transport session remains joined. As of UM version 6.12, TCP-based Topic Resolution will have the same effect. In in a mixed-version UM environment including DROs, it can be difficult to predict when an EOS will be generated for a deleted source.

Be aware that BOS does *not* necessarily indicate end-to-end connectivity between sources and receivers. In a DRO network, where the source is in a different Topic Resolution Domain (TRD) from the receiver, BOS only indicates the establishment of the transport session between the DRO's proxy source and the receiver.

Also, BOS and EOS do not necessarily come in balanced pairs. For example, if a BOS is delivered to a receiver, and then the subscriber application deletes the receiver object, UM does not deliver an EOS to the receiver callback. Also, it is possible to get an EOS event without first getting a BOS. Let's say that a receiver attempts to join an LBT-RM transport session for a source that was just deleted. BOS will not be delivered because there will be no communication of a user or control message. However, after LBT-RM times out the transport session, it will deliver EOS.

Finally, in some versions of UM, EOS represents the end of the entire transport session, so you can have situations where a publisher deletes a source, but the receiver for that topic does not get an EOS for an unbounded period of time. Let's say that the publisher maps sources for topics X, Y, and Z to the same transport session. Let's further say that a subscriber has a receiver for topic Y. It will join the transport session, get a BOS for receiver Y, and will receive messages that the publisher sends to Y. Now let's say that the publisher deletes the source for topic Y, but keeps the sources for topics X and Z. The subscriber may not be informed of the deletion of the source for Y, and can remain subscribed to the transport session. It won't be until the publisher deletes sources X and Z that it will delete the underlying transport session, which can then deliver EOS to the subscriber. However, if using TCP-based Topic Resolution, this same scenario can produce an EOS event on the receiver immediately.

For these reasons and others, Informatica does not recommend using the BOS and EOS events to indicate "connectedness" between a source and a receiver. Instead, Informatica recommends logging the events in your application log file. If something goes wrong, these messages can assist in diagnosing the problem.

For programmers who are tempted to use BOS and EOS events, Informatica recommends instead using the **source_notification_function (receiver)** configuration option. This provides receiving applications a callbacks when a Delivery Controller is created and deleted, and is usually what the programmer *really* wants.

2.16.2 Source Connect and Disconnect Events

There are two source-side events that some applications use but which are not recommended for most use cases:

- **LBM_SRC_EVENT_CONNECT** - Beginning of (transport) Session to receiver.
- **LBM_SRC_EVENT_DISCONNECT** - End of (transport) Session to receiver.

The Connect and Disconnect events are delivered to source on a topic basis, but actually represent the joining and exiting of the underlying Transport Session to a specific receiver.

Note

> The Connect and Disconnect events are *not* available for Transport LBT-RM or Transport LBT-SMX sources.

Note that the Connect and Disconnect source-side events are very similar to the Receiver BOS and EOS Events. However, while BOS and especially EOS have deviated from their pure "transport session" roots, Connect and Disconnect are still purely implemented with respect to the underlying transport session.

Be aware that Connect does *not* necessarily indicate end-to-end connectivity between sources and receivers. In a DRO network, where the source is in a different Topic Resolution Domain (TRD) from the receiver, Connect only indicates the establishment of the transport session between the DRO's proxy receiver and the source. For example, if a source in TRD1 is sending messages to two receivers in TRD2 via a DRO, the source will only receive a single Connect event when the first of the two receivers subscribe. Note also that the IP and port indicated in the Connect event will be for the DRO portal on TRD1.

Also, since Disconnect represents the end of the entire transport session, you can have situations where a subscriber deletes a receiver, but the source for that topic does not get a Disconnect for an unbounded period of time. Let's say that the publisher maps sources for topics X, Y, and Z to the same transport session. Let's further say that a subscriber has a receiver for topic Y and Z. It will join the transport session, and the source will get Connect events for *all three* topics, X, Y, and Z. Now let's say that the subscriber deletes the receiver for topic Y, but keeps the receiver for topic Z. The source will not be informed of the deletion of the receiver for Y, since the transport session continues to be maintained. It won't be until the subscriber deletes both receivers, Y and Z, that it will delete the underlying transport session, which can then deliver Disconnect to the source.

For these reasons and others, Informatica does not recommend using the Connect and Disconnect events to indicate "connectedness". Instead, Informatica recommends logging the events in your application log file. If something goes wrong, these messages can assist in diagnosing the problem.

2.16.3 Source Wakeup Event

The source event **LBM_SRC_EVENT_WAKEUP** is delivered to the application when a non-blocking "send" function is called but returns the **LBM_EWOULDBLOCK** error. When that happens, the message was not sent due to a flow controlling condition. When that flow controlling condition is cleared, UM delivers the wakeup to the source event callback, informing the application that it can resume sending.

The flow controlling conditions are:

- The source uses the TCP (**normal**), IPC (**receiver_paced**), or SMX protocol, and one or more receivers are not reading fast enough. When both the receiver's and sender's buffers are full, a non-blocking "send" will return **LBM_EWOULDBLOCK**. When the slow receiver(s) read more data, the flow control condition is cleared and the wakeup source event is delivered.

- The source uses a UDP-based protocol and the data rate limit is exceeded (see **transport_lbtrm_data_rate_limit (context)** or **transport_lbtru_data_rate_limit (context)**). A non-blocking "send" will return **LBM_EWOULDBLOCK**. When the rate interval expires and the rate limiter adds bandwidth, the flow control condition is cleared and the wakeup source event is delivered.
- The source uses persistence and exceeds a flight size limit (see **ume_flight_size (source)** and **ume_flight_size_bytes (source)**). A non-blocking "send" will return **LBM_EWOULDBLOCK**. When the Store sends stability ACKs to the source and the in-flight message data drops below the limit, the flow control condition is cleared and the wakeup source event is delivered.

In all cases, the wakeup event is delivered by the UM context thread.

However, note that typically a separate thread (created by the application) is calling the non-blocking "send" function. There can be cases where the call to "send" detects the flow control condition, but before it can return **LBM_EWOULDBLOCK**, the context thread clears the condition and delivers the **LBM_SRC_EVENT_WAKEUP** event via the the application's source callback. From the application's point of view, this represents the notification of UM's internal state transitions is reversed, with the wakeup appearing to happen *before* the **LBM_EWOULDBLOCK**. This potential reversal of event notification is a natural consequence of the design of UM and is not considered a bug. The application must be carefully coded to take this race condition into account.

For example, here's a modified excerpt from the **Example lbmsrc.c** program (see similar code in **Java** and **.NET**):

```
int blocked = 0;  /* global state variable. */
...

int handle_src_event(lbm_src_t *src, int event, void *extra_data, void *clientd)
{
    switch (event) {
    case LBM_SRC_EVENT_WAKEUP:
        blocked = 0;
        break;
...
```

The wakeup event indicates that a blocked source is no longer blocked. This example uses a global variable to track the blocked state of the source, so the wakeup event clears it.

Here is the application sending code:

```
/* Assume that the source is blocked. */
blocked = 1;
err = lbm_src_send(src, message, opts->msglen, LBM_SRC_NONBLOCK);
if (err == LBM_FAILURE) {
    if (lbm_errnum() == LBM_EWOULDBLOCK) {
        /* Source is blocked, message did not send. The "lbmsrc.c" example
         * reacts by spin-waiting for "blocked" to be set to zero, and then
         * looping back to re-send the same message.
         * Alternatively, you could simply return to do other work, and
         * re-send the message later when "blocked" is zero.
         * Note that it is possible that "blocked" is already zero at this point.
         */
        }
    }
}
blocked = 0;  /* The message sent successfully; source is not blocked. */
```

The important part of this code is that the global "blocked" flag is set before∗ the call to **lbm_src_send()**. If you try to set it after the send function returns, the wakeup event might execute before your sending thread can set it. This will leave "blocked" set even though the wakeup event has already been delivered.

Note that there should be a separate "blocked" variable for each source object the application creates. This flag would typically be placed in a "clientd" structure instead of being a process-global variable.

Warning

> The above example code is NOT safe for multiple application threads sending to the same source object.

2.16.4 Source Flight Notification Event

The source event **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION** can be delivered to the application when a source is configured for persistence and has set its **ume_flight_size_behavior (source)** to "notify".

As the application sends messages, the flight size increases, and as the Store delivers message stability ACKs, the flight size decreases. As the application sends messages increasing the flight size, if it crosses a flight size limit, the internal flight size state for that source becomes "over" and the **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION** event is delivered with state **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION_STATE_OVER**.

Subsequently, as Store stability acknowledgements are delivered to the source decreasing the flight size, if it goes below the flight size limits, the internal flight size state for that source becomes "under" and the **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION** event is delivered with state **LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION_STATE_UNDER**.

However, these two events are delivered by separate threads. The "OVER" notification is delivered by the application thread calling the "send" function, and the "UNDER" notification is delivered by the UM context receiving the Store's stability ACK. There can be cases where the application will process the two events concurrently, or even in the reverse order that the internal state changed. I.e. the actual order of state transitions might be "UNDER", "OVER", "UNDER, but the application will process the events as "UNDER", "UNDER", "OVER". This potential reversal of event notification is a natural consequence of the design of UM and is not considered a bug. The application must be carefully coded to take this race condition into account.

Here is some example code to demonstrate a possible use pattern:

```
int over_delivered = 0;  /* global. */
int under_delivered = 0;  /* global. */
...

int handle_src_event(lbm_src_t *src, int event, void *extra_data, void *client_data)
{
    switch (event) {
    case LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION:
    {
        lbm_src_event_flight_size_notification_t *notification = (
      lbm_src_event_flight_size_notification_t *)extra_data;
        if (notification->state == LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION_STATE_OVER) {
            over_delivered = 1;
        }
        else {  /* state == LBM_SRC_EVENT_FLIGHT_SIZE_NOTIFICATION_STATE_UNDER */
            under_delivered = 1;
        }
        break;
...
```

This example uses two global state variables to indicate when the two notification events have been delivered.

Here's some code that checks to see if flight size is "over" and wait for it to be "under" before sending:

```
if (over_delivered) {
    while (! under_delivered) {
        /* While waiting for "under", can do other things. */
    }
    /* Both "over" and "under" events were delivered. Reset them so we can send again. */
    over_delivered = 0;
    under_delivered = 0;
}
/* Send next message. */
err = lbm_src_send(src, message, opts->msglen, LBM_SRC_NONBLOCK);
```

This code works because the "over" event is delivered synchronously with the **lbm_src_send()** function. If **lbm_src_send()** exceeds the flight size, UM guarantees that the "over" event will be delivered before the send function returns. This guarantee enables the code to handle the "under" event being delivered before the "over" event.

Note that there should be a separate "over_delivered" and "under_delivered" variables for each source object the application creates. These flags would typically be placed in a "clientd" structure instead of being process-global variables.

Also note that unlike the "wakeup" event (described above), getting an "over" event does not prevent the "send" function from successfully sending the message. The point of setting **ume_flight_size_behavior (source)** to "notify" is to allow the publisher to continue sending in spite of exceeding flight size. But note that doing this can prevent UM from guaranteeing delivery of messages.

Warning

> The above example code is NOT safe for multiple application threads sending to the same source object.

2.17 Rate Controls

For UDP-based communications (LBT-RU, LBT-RM, and Topic Resolution), UM network stability is ensured through the use of rate controls. Without rate controls, sources can send UDP data so fast that the network can be flooded. Using rate controls, the source's bandwidth usage is limited. If the source attempts to exceed its bandwidth allocation, it is slowed down.

Setting the rate controls properly requires some planning; see `Topics in High Performance Messaging, Group Rate Control` for details.

Ultra Messaging's rate limiter algorithms are based on dividing time into intervals (configurable), and only allowing a certain number of bits of data to be sent during each interval. That number is divided by the number of intervals per second. For example, a limit of 1,000,000 bps and an interval of 100 ms results in the limiter allowing 100,000 bits to be sent during each interval. Dividing by 8 to get bytes gives 12,500 bytes per interval.

Data are not sent over a network as individual bytes, but rather are grouped into datagrams. Since it is not possible to send only part of a datagram, the rate limiter algorithm needs to decide what to do if an outgoing datagram would exceed the number of bits allowed during the current time interval. The data transport rate limiter algorithm, for LBT-RM and LBT-RU, differs from the Topic Resolution rate limiter algorithm.

2.17.1 Transport Rate Control

With data transport, if an application sends a message and the outgoing datagram would exceed the number of bits allowed during the current time interval, that datagram is queued and the transport type is put into a "blocked" state in the current context. Note that success is returned to the application, even though the datagram has not yet been transmitted.

However, any subsequent sends within the same time interval will not queue, but instead will either block (for blocking sends), or return the error **LBM_EWOULDBLOCK** (for non-blocking sends). When the time interval expires, the context thread will refresh the number of allowable bits, send the queued datagram, and unblock the transport type.

Note that for very small settings of transport rate limit, the end-of-interval refresh of allowable bits may still not be enough to send a queued full datagram. In that case, the datagram will remain on the queue for additional intervals to pass, until enough bits have accumulated to send the queued datagram. However, it would be very unusual for a transport rate limit to be set that small.

Transport Rate Control Configuration

Configuration parameters of interest are:

- **transport_lbtrm_rate_interval (context)**
- **transport_lbtrm_data_rate_limit (context)**
- **transport_lbtrm_retransmit_rate_limit (context)**
- **transport_lbtru_rate_interval (context)**
- **transport_lbtru_data_rate_limit (context)**
- **transport_lbtru_retransmit_rate_limit (context)**

LBM_EWOULDBLOCK

When non-blocking sends are done (LBM_SRC_NONBLOCK), the send can fail with an error number of LBM_EWOULDBLOCK. This means you have exceeded the rate limit and must wait before sending the message.

Most applications don't need to handle this because they use blocking sends. Instead of returning a failure when the rate limit is exceeded, the send function sleeps until the transport becomes unblocked again, at which point the UM context thread wakes up the sleeping send call. However, there are times that blocking behavior is not permitted. For example, if the application is running as a UM Context thread callback (timer, receiver event, source event, etc), then only non-blocking sends are allowed.

To assist the application in retrying the message send, the UM context thread will deliver the LBM_SRC_EVENT_WAKEUP source event to the application. (This only happens if a non-blocking send was tried and failed with LBM_EWOULDBLOCK.) The application can use that event to re-try sending the message.

Alternatively, exceeding the rate limit can be avoided by increasing the configured rate limit and/or rate interval. This can reduce latency, but also increases the risk of packet loss.

The rate limit controls how many bits can be transmitted during a given second of time. The rate interval controls the "shape" of the transmissions over the second of time.

Adjusting the Rate Interval

A long rate interval will allow intense bursts of traffic to be sent, which can overload switch port queues or NIC ring buffers, leading to packet loss. A short rate interval will reduce the intensity of the bursts, spreading the burst over a longer time, but does so by introducing short delays in the transmissions, which increases message latency.

The LBT-RM default rate interval of 10 milliseconds was chosen to be reasonably "safe" in terms of avoiding loss, at the expense of possible transmission delays. Latency-sensitive applications may require a larger value, but increases the risk of switch or NIC loss.

There is no analytical way of choosing an optimal value for the rate interval. Latency-sensitive users should test different values with intense bursts to find the largest value which avoids packet loss.

2.17.2 Topic Resolution Rate Control

With UDP-based Topic Resolution ("TR"), the algorithm acts differently. It is designed to allow at least one datagram per time interval, and is allowed to exceed the rate limit by at most one topic's worth. Thus, the TR rate limiter value should only be considered a "reasonably accurate" approximation.

This approximation can seem very inaccurate at very small rate limits. As an extreme example, suppose that a user configures the **sustaining advertisement rate limiter** to 1 bit per second. Since the TR rate limiter allows at least one Advertisement (TIR) to be sent per interval, and a TIR of a 240-character topic creates a datagram about 400 bytes long (exact size depends on user options), ten of those per second is 32,000 bits, which is over 3 million percent of the desired rate. This sounds extreme, but understand that this works out to only 10 packets per second, a trivial load for modern networks. In practice, the minimum *effective* rate limit works out to be one datagram per interval.

For details of Topic Resolution, see Topic Resolution Description.

2.18 Operational Statistics

UM maintains a variety of transport-level statistics which gives a real-time snapshot of UM's internal handling. For example, it gives counts for transport messages transferred, bytes transferred, retransmissions requested, unrecoverable loss, etc.

The UM monitoring API provides framework to allow the convenient gathering and transmission of UM statistics to a central monitoring point. See **Monitoring** for more information.

2.19 Immediate Messaging

UM has an alternate form of messaging that deviates from the normal publish/subscribe paradigm. An "immediate message" can be sent and received without the need for topic resolution. Note that immediate messaging is less efficient than normal source-based messaging and will not provide the same low latency and high throughput.

There are two forms of immediate messaging:

- Unicast Immediate Messaging (UIM) - TCP-based point-to-point.
- Multicast Immediate Messaging (MIM) - flooding-based multicast; use rarely.

Chapter 3

UM Objects

Many UM documents use the term object. Be aware that with the C API, they do not refer to formal class-based objects as supported by C++. The term is used here in an informal sense to denote an entity that can be created, used, and (usually) deleted, has functionality and data associated with it, and is managed through the API. The handle that is used to refer to an object is usually implemented as a pointer to a data structure (defined in **lbm.h**), but the internal structure of an object is said to be opaque, meaning that application code should not read or write the structure directly.

However, the UM Java and .NET APIs *are* formally object oriented, with Java/C# class-based objects. See the **UM Java API** and the `UM .NET API` documentation.

The UM API is designed around a set of UM objects. The most-often used object types are:

- Context - a fairly "heavy-weight" parent object for most other UM objects. Also represents an "instance" of UM, with all foreground and background functions.
- Source - a publisher creates one or more Source objects, one per topic to be published.
- Receiver - a subscriber creates one or more Receivers, one per topic that is subscribed.
- Wildcard Receiver - a subscriber can optionally create Wildcard Receivers to subscribe to patterns that can match multiple topics.
- Hot Failover Receiver - a specialized type of Receiver object that allows for redundant publishing.
- HFX Receiver - a specialized type of Hot Failover Receiver object that allows redundant sources across multiple Context objects. (Note: HFX is deprecated.)
- Event Queue - an active object (requires one or more threads) which is used to queue and deliver UM events (including received messages) to the application using a separate thread. Event queues are normally not required; by default UM delivers events using the Context's own thread. But there are circumstances where it is useful to transfer the received message or event to an independent thread for delivery.

A typical UM application program manages UM objects following certain ordering rules:

1. If desired, Event Queues are typically created first. (Most UM applications do not use Event Queues.)
2. Create a Context object. A typical application creates only one context, but there are some specialized use cases in which a small number of contexts (typically less than 5) are useful.
3. Create one or more Source and Receiver objects. It is not unusual for an application to create thousands of sources or receivers.
4. The application's main processing phase consists of publishing messages using the Source objects, and receiving messages using the Receiver objects. UM is most efficiently used if those Source and Receiver objects are created during initialization. Dynamic object creation during normal operation is possible, but can require special coding.

5. When the application is ready to shut down, Sources and Receivers should be deleted.
6. Contexts are deleted after all Sources and Receivers are deleted. Note that if the context uses Sequential Mode, the event processing thread should be unblocked and joined prior to Context deletion.
7. Event Queues, if used, are deleted last. Note that the Event Queue dispatching thread(s) should be unblocked and joined prior to deletion of the Event Queue.

Note that it is very important to enforce the above deletion order. For example, it can lead to crashes if you delete a context while it still has active Sources or Receivers associated with it. Similarly, crashes can result if you delete an event queue while it still has active Contexts associated with it.

Also, note that deletion of Source objects can affect the reliability of message delivery. UM Receivers are designed to detect lost packets and request retransmission. However, once a Source object is deleted, it can no longer fulfill retransmission requests. It is usually best for an application to delay a few seconds after sending its last messages before deleting the Sources. This is especially important if Implicit Batching is used since outgoing messages might be held in the Batcher.

3.1 Context Object

A UM context object conceptually is an environment in which UM runs. An application creates a context, typically during initialization, and uses it for most other UM operations. In the process of creating the context, UM normally starts an independent thread (the context thread) to do the necessary background processing such as the following:

- Topic resolution
- Enforce rate controls for sending messages
- Manage timers
- Manage state
- Implement UM protocols
- Manage Transport Sessions

You create a context with **lbm_context_create()**. When an application is finished with the context (no more message passing needed), it should delete the context by calling **lbm_context_delete()**.

Warning

> Before deleting a context, you must first delete all objects contained within that context (sources, receivers, wildcard receivers). See Deleting UM Objects.

Your application can give a context a name, which are optional but should be unique across your UM network. You set a context name before calling **lbm_context_create()** in the following ways:

- If you are using XML UM configuration files, call **lbm_context_attr_set_from_xml()** or **lbm_context_attr_create_from_xml()** and set the name in the **context_name (context)** parameter.
- If you are using plain text UM configuration files, call **lbm_context_attr_setopt()** and specify **context_name (context)** as the optname and the context's name as the optval. Don't forget to set the optlen.
- Create a plain text UM configuration file with the option **context_name (context)** set to the name of the context.

Context names are optional but should be unique within a process. UM does not enforce uniqueness, rather issues a log warning if it encounters duplicate context names. Application context names are only used to load template and individual option values within an XML UM configuration file.

One of the more important functions of a context is to hold configuration information that is of context scope. See the `UM Configuration Guide` for options that are of context scope.

Most UM applications create a single context. However, there are some specialized circumstances where an application would create multiple contexts. For example, with appropriate configuration options, two contexts can provide separate topic name spaces. Also, multiple contexts can be used to portion available bandwidth across topic sub-spaces (in effect allocating more bandwidth to high-priority topics).

Attention

> Regardless of the number of contexts created by your application, a good practice is to keep them open throughout the life of your application. Do not close them until you close the application.

3.2 Topic Object

A UM topic object is conceptually very simple; it is little more than a container for a string (the topic name). However, UM uses the topic object to hold a variety of state information used by UM for internal processing. It is conceptually contained within a context. Topic objects are used by applications in the creation of sources and receivers.

Technically, the user's application does not create or delete topic objects. Their management is handled internally by UM, as needed. The application uses APIs to gain access to topic objects. A publishing application calls **lbm↵_src_topic_alloc()** to get a reference to a topic object that it intends to use for creation of a Source Object. A subscribing application calls **lbm_rcv_topic_lookup()** to get a reference to a topic object that it intends to use for creation of a Receiver Object.

The application does not need to explicitly tell UM when it no longer needs the topic object. The application's reference can simply be discarded.

3.3 Source Object

A UM source object is used to send messages to the topic that it is bound to. It is conceptually contained within a context.

You create a source object by calling **lbm_src_create()**. One of its parameters is a Topic Object. A source object can be bound to only one topic. The application is responsible for deleting a source object when it is no longer needed by calling **lbm_src_delete()**. See Deleting UM Objects.

3.3.1 Source String

Every source that a publishing application creates has associated with it a unique "source string". Note that if multiple publishing UM contexts (applications) create sources for the same topic, each context's source will have its own unique source string. Similarly, if one publishing UM context (application) creates multiple sources for different topics, each topic's source will have its own unique source string. So a source string identifies one specific instance of a topic within a UM context.

The source string is used in a few different ways in the UM API, for example to identify which Transport Session to retrieve statistics for in **lbm_rcv_retrieve_transport_stats()**. The source string is made available to the application in several callbacks, for example **lbm_src_notify_function_cb**, or the "source" field of **lbm_msg_t_stct** of a received message. See also Sending to Sources.

The format of a source string depends on the transport type:

- TCP:src_ip:src_port:session_id[topic_idx]
  example: `TCP:10.29.3.88:45789:f1789bcc[1539853954]`
- LBTRM:src_ip:src_port:session_id:mc_group:dest_port[topic_idx]
  example: `LBTRM:10.29.3.88:14390:e0679abb:231.13.13.13:14400[1539853954]`
- LBT-RU:src_ip:src_port:session_id[topic_idx]
  example: `LBT-RU:10.29.3.88:14382:263170a3[1539853954]`
- LBT-IPC:session_id:transport_id[topic_idx]
  example: `LBT-IPC:6481f8d4:20000[1539853954]`
- LBT-SMX:session_id:transport_id[topic_idx]
  example: `LBT-SMX:6481f8d4:20000[1539853954]`
- BROKER
  example: `BROKER`

Please note that the topic index field (topic_idx) may or may not be included, depending on the context in which it is presented, your version of UM, and the setting for configuration option **source_includes_topic_index (context)**. For example, a receiver callback probably will include the topic index since the callback is specific to a topic, while **Monitoring** output will not, since it is referring to a transport session, not an individual topic.

See also **lbm_transport_source_format()** and **lbm_transport_source_parse()**.

3.3.2 Source Strings in a Routed Network

In a UM network consisting of multiple Topic Resolution Domains (TRDs) connected by DROs, a given source will be uniquely identifiable within a TRD by its source string. However, that same source will have different source strings in different TRDs. For receivers in the same TRD as the source, the source string will refer to the source. But in remote TRDs, that same source's source string will refer to the *proxy source* of the DRO on the shortest path to the source. The IP information contained in the source string will refer to the DRO.

This can lead to a situation where multiple originating sources located elsewhere in the UM network will have source strings with the same IP information in a given TRD. They will differ by Topic Index number, but even that topic index will be different in different TRDs.

3.3.3 Source Configuration and Transport Sessions

As with contexts, a source holds configuration information that is of source scope. This includes network options, operational options and reliability options for LBT-RU and LBT-RM. For example, each source can use a different transport and would therefore configure a different network address to which to send topic messages. See the `UM Configuration Guide` for source configuration options.

As stated in UM Transports, many topics (and therefore sources) can be mapped to a single transport. Many of the configuration options for sources actually control or influence Transport Session activity. If many sources are sending topic messages over a single Transport Session (TCP, LBT-RU or LBT-RM), UM only uses the configuration options for the first source assigned to the transport.

For example, if the first source to use a LBT-RM Transport Session sets the **transport_lbtrm_nak_generation_interval (receiver)** to 24 MB and the second source sets the same option to 2 MB, UM assigns 24 MB to the Transport Session's **transport_lbtrm_nak_generation_interval (receiver)**.

The UM Configuration Guide identifies the source configuration options that may be ignored when UM assigns the source to an existing Transport Session. Log file warnings also appear when UM ignores source configuration options.

3.4 Receiver Object

A UM receiver object is used to receive messages from the topic that it is bound to. It is conceptually contained within a context. Messages are delivered to the application by an application callback function, specified when the receiver object is created.

You create a receiver object by calling **lbm_rcv_create()**. One of its parameters is a Topic Object. A receiver object can be bound to only one topic. The application is responsible for deleting a receiver object when it is no longer needed by calling **lbm_rcv_delete()**. See Deleting UM Objects.

Multiple receiver objects can be created for the same topic within a single context, which can be used to trigger multiple delivery callbacks when messages arrive for that topic.

3.4.1 Receiver Configuration and Transport Sessions

A receiver holds configuration information that is of receiver scope. This includes network options, operational options and reliability options for LBT-RU and LBT-RM. See the UM Configuration Guide for receiver configuration options.

As stated above in Source Configuration and Transport Sessions, multiple topics (and therefore receivers) can be mapped to a single transport. As with source configuration options, many receiver configuration options control or influence Transport Session activity. If multiple receivers are receiving topic messages over a single Transport Session (TCP, LBT-RU or LBT-RM), UM only uses the configuration options for the first receiver assigned to the transport.

For example, if the first receiver to use a LBT-RM Transport Session sets **transport_lbtrm_nak_generation_interval (receiver)** to 10 seconds, that value is applied to the Transport Session. If a second receiver using the same transport session sets the same option to 2 seconds, that value is ignored.

The UM Configuration Guide identifies the receiver configuration options that may be ignored when UM assigns the receiver to an existing Transport Session. Log file warnings also appear when UM ignores receiver configuration options.

3.4.2 UM Wildcard Receivers

You create a wildcard receiver object by calling **lbm_wildcard_rcv_create()**. Instead of a topic object, the caller supplies a pattern which UM uses to match multiple topics. Because the application does not explicitly lookup the topics, UM passes the topic attribute into **lbm_wildcard_rcv_create()** so that it can set options. Also, wildcard receivers have their own set of options, such as pattern type. The application is responsible for deleting a wildcard receiver object when it is no longer needed by calling **lbm_wildcard_rcv_delete()**.

The wildcard pattern supplied for matching is a PCRE regular expression that Perl recognizes. See http://perldoc.perl.org/perlrequick.html for details about PCRE.

Note

> Ultra Messaging has deprecated two other wildcard receiver pattern types, regex POSIX extended regular expressions and appcb application callback, as of UM Version 6.1. Only PCRE is supported.

Be aware that some platforms may not support all of the regular expression wildcard types. For example, UM does not support the use of Unicode PCRE characters in wildcard receiver patterns on any system that communicates with a HP-UX or AIX system. See the Informatica Knowledge Base article, `Platform-Specific Dependencies` for details.

For examples of wildcard usage, see **Example lbmwrcv.c**, **Example lbmwrcv.java**, and **Example lbmwrcv.cs**.

For more information on wildcard receivers, see Wildcard Receiver Topic Resolution, and **Wildcard Receiver Options**.

TIBCO users see the Informatica Knowledge Base articles, `Wildcard topic regular expressions and SmartSockets wildcards` and `Wildcard topic regular expressions and Rendezvous wildcards`.

Overlapping Receivers

Suppose an application creates three receivers:

1. Wildcard receiver1: "^example[0-9]$"
2. Single-topic receiver: "example1".
3. Wildcard receiver2: "^[a-z]*1$"

A source for topic "example1" will match all three receivers. Each receiver's object's "receiver callback" will be invoked sequentially for each received message. However, be aware that these are simply multiple callbacks from a single underlying UM receiver; they are not independent.

This becomes significant if different "receiver" scoped configuration options are specified for the different receiver objects, a usage that Informatica recommends against. Only one of the receiver object's configuration is used to create the underlying receiver; the others are ignored. In the above example, it is not necessarily the first receiver object which applies its configuration. If, for example, the source is not yet discovered until later, UM does not define which of the above three receiver objects will be used.

Informatica recommends users always use the same configuration options when creating multiple receiver objects that overlap.

3.4.3 Transport Services Provider Object

The Transport Services Provider object ("XSP") is introduced with UM version 6.11 and beyond to manage sockets, threads, and other receive-side resources associated with subscribed Transport Sessions. The primary purpose for an XSP object is to allow the programmer to control the threading of received messages, based on the Transport Sessions of those messages.

For more information on XSP, see Transport Services Provider (XSP).

3.4.4 UM Hot Failover Across Contexts Objects

Hot Failover Across Contexts objects ("HFX") provide a form of hot failover that can operate across multiple network interfaces.

Note that the HFX feature is deprecated.

For more information, see Hot Failover Across Multiple Contexts (HFX).

3.5 Event Queue Object

Most UM events, like received messages, are delivered to the application via event handler callbacks, like a receiver callback. A UM "event queue" object is a queue for moving execution of a UM callbacks to a different thread - the event queue dispatch thread.

Note that unlike other UM objects, event queues are not owned by a context. An event queue is a top-level object, and can be associated with contexts, sources, and/or receivers.

Warning

> Before deleting an event queue, you must first delete all objects associated with it (contexts, sources, receivers, timers). See Deleting UM Objects.

By default, UM events will be delivered from a variety of different threads, frequently a context or XSP thread. Context/XSP thread callbacks are the most efficient form of event delivery, but place restrictions on your callback code. For example:

- The application function is not allowed to make certain API calls (mostly having to do with creating or deleting UM objects).
- The application function must execute very quickly without blocking.
- The application does not have control over when the callback executes. For example, events might be delivered concurrently from different application and context/XSP threads.

For these reasons, you might want to transfer handling of UM events to a different thread. This can impose strict serialization and remove restrictions.

You could make use of your own queue for this, perhaps a standard library queue (although be careful of multiple threads enqueuing on the same queue), or perhaps `LMAX's "Disruptor"`. Or you can use the UM event queue.

An advantage of the UM event queue is that it retains all of the semantics of the UM callback. The callback code is structured the same, regardless of whether it is called from a context/XSP thread or from an event queue dispatch thread. A disadvantage of the UM event queue is efficiency: it makes a kernel call for each enqueue and each dequeue.

UM event queues are unbounded, non-blocking queues and provide the following features:

- Queue length monitoring. Informatica strongly recommends using the event queue monitor callback to warn if the configured **size** or **latency** thresholds are exceeded. (Note: exceeding a threshold does not prevent new events from being enqueued.) See Event Queue Monitor for more information.
- The application callback function has no UM API restrictions.
- Your application can control exactly when UM delivers queued events with **lbm_event_dispatch()**. And you can have control return to your application either when specifically asked to do so (by calling **lbm_event_dispatch_unblock()**), or optionally when there are no events left to deliver.
- Your application can take advantage of parallel processing on multiple processor hardware since UM processes asynchronously on a separate thread from your application's processing of received messages. By using multiple application threads to dispatch an event queue, or by using multiple event queues, each with its own dispatch thread, your application can further increase parallelism.

You create an UM event queue in the C API by calling **lbm_event_queue_create()**. When finished with an event queue, delete it by calling **lbm_event_queue_delete()**.

In the Java API and the .NET API, use the **LBMEventQueue** class.

See **Event Queue Options** for configuration options related to event queues.

3.5.1 Using an Event Queue

To use an Event Queue, an application typically performs the following actions:

1. Create the Event Queue using **lbm_event_queue_create()**.

   ```
   lbm_event_queue_t *evq;
   err = lbm_event_queue_create(&evq, NULL, NULL, NULL);
   ```

2. Create a new thread of execution to be the dispatch thread. This new thread should call **lbm_event_dispatch()** in a loop.

   ```
   evq_running = 1;
   while (evq_running) {
     err = lbm_event_dispatch(evq, LBM_EVENT_QUEUE_BLOCK);
     if (err == LBM_FAILURE) { ... handle error ... }
   }
   /* Exit the thread (OS-dependent). */
   ```

   Note that the return value should be compared to **LBM_FAILURE** (-1), and not the normal success value of 0. This is because on success, **lbm_event_dispatch()** returns the number of events that were dispatched during its execution.

3. Create other UM objects whose events you want processed by the event queue. For example, creating a UM Receiver object with the Event Queue will call your message receiver callback through the event queue, using your dispatch thread.

   ```
   lbm_rcv_t *rcv;
   err = lbm_rcv_create(&rcv, ctx, topic, app_rcv_callback, NULL, evq);
   ```

   From this point, your application receiver callback function will be called from the dispatch loop for received messages and other receiver events.

4. When it is time to shut down the program, the UM objects that refer to the event queue must first be deleted.

   ```
   err = lbm_rcv_delete(rcv);
   ```

   Remember that an event queue might be handling events for many UM objects; they must all be deleted prior to deleting the event queue.

5. Now shut down dispatching the event queue.

   ```
   evq_running = 0;
   err = lbm_event_dispatch_unblock(evq);
   /* "Join" the dispatch thread (OS-dependent). */
   ```

   The unblock forces the **lbm_event_dispatch()** function to return. Typically at that point, the dispatch thread is "joined", which blocks until the dispatch thread exits.

6. Delete the event queue.

   ```
   err = lbm_event_queue_delete(evq);
   ```

Here are some lesser used event queue APIs:

- **lbm_event_queue_size()** - number of events in the queue.
- **lbm_event_queue_shutdown()** - purge events from the queue (risky).
- **lbm_event_queue_retrieve_stats()** - retrieve statistics counters.
- **lbm_event_queue_reset_stats()** - reset statistics counters.

3.5.2 "Deleting an Event Queue"

Before you can delete an event queue, you must ensure that your application's dispatch thread is no longer running it. Here is the typical order of operation for deleting an event queue:

1. **Unblock the event queue** (**C API** / **Java and .NET API**). This causes the application's dispatch thread to return from the dispatch function (**C API** / **Java and .NET API**).
2. **Exit the application's dispatch thread**, typically with a thread join function.
3. **Delete the event queue** (**C API** / **Java and .NET API**).

3.5.3 Event Queue Efficiency

A UM event queue introduces a kernel call with each enqueue and dequeue. Kernel calls are considered costly for very low latency or high throughput applications.

Compared with a lockless, busy waiting queue like `LMAX's "Disruptor"`, the UM event queue will have a lower maximum sustainable throughput and higher latency. The latency penalty will be especially apparent at low message rates where the dispatch thread is put to sleep waiting for new events.

The performance of the UM event queue can be improved by using Receive-Side Batching and by polling the event queue in a tight loop (see Event Queue Timeout).

3.5.4 Event Queue Timeout

The second parameter passed to **lbm_event_dispatch()** is a timeout. There are two special values:

- **LBM_EVENT_QUEUE_BLOCK** - no timeout. Do not return until **lbm_event_dispatch_unblock()** is called.
- **LBM_EVENT_QUEUE_POLL** - no wait. If there are one or more events in the queue, process exactly one of them and return. Otherwise, return immediately without waiting.

Any other value specifies a timeout in milliseconds.

However, this timeout does not necessarily limit the time spent waiting inside the dispatch function. The purpose of the timeout is to set a *minimum* time that the dispatch function will process events, not a maximum.

The implementation of the event queue uses an unbounded wait for incoming events. When an event is delivered to the queue, the dispatch function wakes up and processes the event (calls the appropriate application callback). Then the dispatch function checks the time to see if the timeout has been exceeded. If so, the dispatch function returns. Otherwise, it waits for the next event.

However, suppose that no further events are delivered to the event queue. In this case, the dispatch function will wait without bound. The timeout parameter will not cause the dispatch function to stop waiting.

If the application needs an upper limit to the time spent dispatching, the timeout can be combined with the use of an external timer that calls **lbm_event_dispatch_unblock()** when the maximum time has expired. UM's timer system may be used by calling **lbm_schedule_timer()**.

3.5.5 Event Queue Monitor

The UM event queue is unbounded, meaning that it doesn't have a fixed maximum size. Rather, if the incoming event (e.g. message) rate exceeds the callback processing rate, the event queue will grow without bound, eventually consuming the maximum available memory, usually crashing the program.

The only way to prevent this unbounded memory growth is to ensure that your event handler callback can keep up with the incoming event rate. However, this is usually impossible to guarantee, so it is important for applications to monitor queue growth and at least raise an alert if it exceeds some threshold.

The event queue size can change very quickly during traffic bursts. **Automatic Monitoring** is not a good solution for detecting these rapid spikes in queue length. UM has a special form of monitoring designed specifically for event queues: a monitoring callback.

You set the size threshold, in number of events, using **queue_size_warning (event_queue)**. You can also set a latency threshold, in microseconds that events are waiting in the event queue, using **queue_delay_warning (event_queue)**. It is also possible to be notified every time an event is added to the event queue, using **queue_enqueue_notification (event_queue)** (no information is available about the event).

UM tests the queue against the thresholds during the process of dispatching events. When the dispatch thread dequeues an event, it checks the size of the queue and the amount of time the dequeued event spent in the queue. If either or both exceed their thresholds, the monitor callback is called. If there is a burst of traffic, the size threshold can be exceeded by many events. In that case, the monitor callback will be called repeatedly for each dequeued event that exceeds a threshold.

Informatica strongly recommends minimizing the monitor callback to perform as little work as possible. Remember that the event queue grows when the event handling callback cannot keep up with the incoming event rate. Given that the monitor callback is called by the dispatch thread, its execution time is added to your event handler, slowing it down even further. We recommend setting a "high water mark" of the maximum detected queue size, and letting a separate thread periodically check that high water mark and raise an alert.

Note that if an application event handling callback were to deadlock and never return, the dispatch thread would be effectively halted, and the monitor callback would never be called. The monitor callback cannot be used to detect a hung application.

C Code Example:

```
/* Define monitor callback. */
int evq_monitor(lbm_event_queue_t *evq, int event, size_t evq_size,
                lbm_ulong_t evq_delay_usec, void *clientd)
{
  /* The "event" parameter is sometimes used as a bitmap, and other times
   * used as an absolute value. */
  if ((event == LBM_EVENT_QUEUE_ENQUEUE_NOTIFICATION)
       && (evq_size == 1) && (evq_delay_usec == 0)) {
    /* This is an ENQUEUE notification. No info available on the event. */
    ...
  }
  else {
    /* "event" is a bitmap; one or both conditions might be active. */
    if (event & LBM_EVENT_QUEUE_SIZE_WARNING) {
      /* The size threshold is exceeded for the next event to be dispatched.
       * There are evq_size events in the queue. */
      ...
    }
    if (event & LBM_EVENT_QUEUE_DELAY_WARNING) {
      /* The delay threshold is exceeded for the next event to be dispatched.
       * The next event to be dispatched waited in the queue for evq_delay_usec */
      ...
    }
  }
}  /* evq_monitor */

  ...
  /* Initialization code, create event queue. */
  lbm_event_queue_t *evq;
  ...
  err = lbm_event_queue_create(&evq, evq_monitor, NULL, NULL);
```

Java and .NET Code Example:

```
/* Define monitor callback. */
class MyEventQueue extends LBMEventQueue implements LBMEventQueueCallback
{
  public MyEventQueue() throws LBMException
  {
      super();
      addMonitor(this);
  }

  public void monitor(Object cbArg, int event, int evqSize, long evqDelayUsec)
  {
    /* The "event" parameter is sometimes used as a bitmap, and other times
     * used as an absolute value. */
    if ((event == LBM.EVENT_QUEUE_ENQUEUE_NOTIFICATION)
         && (evqSize == 1) && (evqDelayUsec == 0)) {
      /* This is an ENQUEUE notification. No info available on the event. */
      ...
    }
    else {
      /* "event" is a bitmap; one or both conditions might be active. */
      if (event & LBM.EVENT_QUEUE_SIZE_WARNING) {
        /* The size threshold is exceeded for the next event to be dispatched. */
        ...
      }
      if (event & LBM.EVENT_QUEUE_DELAY_WARNING) {
        /* The delay threshold is exceeded for the next event to be dispatched. */
        ...
      }
    }
  }  /* monitor */
}
...

  /* Initialization, create event queue. */
  MyEventQueue evq = new MyEventQueue();
```

3.6 Message Object

When an application subscribes to a topic to which publishers are sending messages, the received messages are delivered to the application by an application callback function (see Event Delivery). One of the parameters that UM passes to the application callback is a message object. This object gives the application access to the content of the message, as well as some metadata about the message, such as the topic.

Unlike other objects described above, the user does not create these message objects by API call. UM creates and initializes the objects internally.

The default life-span of a message object is different between C and Java or .NET.

3.6.1 Message Object Deletion

C API

In C, by default, the message object is deleted when the receiver callback returns. No action is necessary by the application to trigger that deletion.

See Message Reception for details, including code examples.

Java or .NET API

In Java or .NET, the passed-in message object is *not* automatically deleted when the receiver application callback returns. Instead, the message object is fully deleted only when all references to the object are lost and the garbage collector reclaims the object.

However, applications which allow this kind of garbage buildup and collection usually suffer from large latency outliers (jitter), and while garbage collection can be tuned to minimize its impact, it is usually recommended that latency-sensitive applications manage their objects more carefully to prevent creation of garbage. See Zero Object Delivery.

Also, there are some UM features in which specific actions are triggered by the deletion of messages, and the application designer usually wants to control when those actions are performed (for example, **Persistence Message Consumption**).

For these reasons, Java and .NET developers are strongly advised to explicitly dispose of a message object when the application is finished with it. It does this by calling the "dispose()" method of the message object. In the simple case, this should be done in the receiver application callback just before returning.

See Message Reception for details, including code examples.

3.6.2 Message Object Retention

Some applications are designed to process received messages in ways that cannot be completed by the time the receiver callback returns. In these cases, the application must extend the life span of the message object beyond the return from the receiver application callback. This is called "message retention".

Note that message retention prevents the recycling of the UM receive buffer in the UM library. See Receive Buffer Recycling.

C API

To prevent automatic deletion of the message object when the receiver application callback returns, the callback must call **lbm_msg_retain()**. This allows the application to transfer the message object to another thread, work queue, or control flow.

When a received message is retained, it becomes the application's responsibility to delete the message explicitly by calling **lbm_msg_delete()**. Failure to delete retained messages can lead to unbounded memory growth.

See Message Reception for details, including code examples.

Java or .NET

The receiver application callback typically calls the "promote()" method of the message object prior to returning. See **Retaining Messages**.

See Message Reception for details, including code examples.

3.7 Attributes Object

An attribute object is used to programmatically configure other UM objects. Their use is usually optional; omitting them results in default configurations (potentially overridden by configuration files). See **Configuration Overview** for details.

However, there is a class of configuration options that require the use of an attribute object: configuring application callbacks. With these options, you are setting a value that includes a pointer to a C function. This cannot be done from a configuration file. For example, see **source_notification_function (receiver)**.

See **Attributes Objects** for more details on the types, creation, use, and deletion of attributes objects.

3.8 UM Timers

UM also provides a timer function whereby the application can schedule callbacks to be made some number of milliseconds in the future. When a given timer expires, its callback is made. A timer can also be canceled prior to expiration.

These timers are not designed like other UM objects. For example, in the C API, you don't "create" and "delete" them, you "schedule" and optionally "cancel" them. In the Java and .NET APIs, you do instantiate a timer object in the normal way, but you don't "close" them when done.

Warning

> UM timers are not designed for high accuracy. For example, to improve internal efficiency, UM will sometimes expire a timer up to 3 milliseconds early. Also, depending on other work being done, a timer can expire but its callback execution can be delayed an undefined amount of time.

Main timer API functions:

- **Schedule** - **C API** / **Java and .NET API**.
- **Cancel** - **C API** / **Java and .NET API**.

Chapter 4

Security Considerations

UM should generally be used in secure networks where unauthorized users are unable to access UM components and applications. A UM network can be made secure through the use of certificate-based encryption (see Encrypted TCP), but this increases message delivery latency and reduces maximum possible throughput.

In particular, the use of UDP-based protocols (LBT-RM and LBT-RU) cannot secured in the same way that TCP can. In a system that uses UDP-based protocols, there is no mechanism in UM to prevent unauthorized applications to be deployed which can then subscribe and publish topics with complete freedom.

4.1 Webmon Security

Of special interest is the use of simple web-based monitoring of the UM daemons: Store, DRO, SRS. UM does not provide any sort of authentication or authorization for the daemons web pages.

Be aware that the use of UM daemon web-based monitoring pages does place a load on the daemon being monitored. For some pages, that load can be significant. An unauthorized user who rapidly accesses pages can disrupt the normal operation of the daemon, potentially leading to message loss.

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 daemon web monitors. See:

Daemon	Configuration Element to Disable Web Monitor
Store	<**web-monitor**>
DRO	<**web-monitor**>
SRS	<debug-monitor>

Chapter 5

Configuration Introduction

UM is designed to be highly configurable. Configuration options are generally given default values to support good performance over a wide variety of use cases, but for users who demand the highest levels of performance, many configuration options allow for optimization.

Where practical, UM's design philosophy is to offer new features that can be enabled via configuration only, without requiring changes to the application source code. This can provide higher performance and reliability just by upgrading, without needing to rebuild the application.

Broadly speaking, there are two kinds of components in a UM-based distributed system that need to be configured:

1. Application programs, linked with the UM library,
2. Informatica daemons (service programs that run in the background), which provide services to the applications.

Application programs are written by users and call UM API functions contained within the UM library. The UM library can be configured using a number of different methods described in **Configuration Overview**.

Informatica daemons include:

- SRS (Stateful Resolver Service) - provides TCP-based topic resolution services to other components. See SRS Configuration File for configuration details.
- lbmrd (Resolver Daemon) - provides unicast UDP-based topic resolution services to other components. See lbmrd Configuration File for configuration details.
- Stored - provides persistence services. See **Configuration Reference for Umestored** for configuration details.
- DRO - routes messages between Topic Resolution Domains. See **DRO Configuration Reference** for configuration details.
- ActiveMQ Broker - provides general queuing services and JMS API. See `ActiveMQ Xml Configuration` for configuration details.

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`
- Informatica daemons (e.g. SRS, Store, DRO) are configured using program-specific configuration files in XML format.
- Informatica daemons (e.g. SRS, Store, DRO) also internally create UM objects (contexts, sources, receivers) using the UM library. Those objects must also be configured using one or more LBM configuration files.

Chapter 6

Transport Types

6.1 Transport TCP

The TCP UM transport uses normal TCP connections to send messages from sources to receivers. This is the default transport when it's not explicitly set. TCP is a good choice when:

LBT-TCP's Transport Pacing can be either source-paced or receiver-paced. See **transport_tcp_multiple_receiver_behavior (source)**.

- Flow control is desired. For example, when one or more receivers cannot keep up, you wish to slow down the source. But see TCP Flow Control Restrictions.
- Equal bandwidth sharing with other TCP traffic is desired. I.e. when it is desired that the source slow down when general network traffic becomes heavy.
- There are few receivers listening to each topic. Multiple receivers for a topic requires multiple transmissions of each message, which places a scaling burden on the source machine and the network.
- The application is not sensitive to latency. Use of TCP as a messaging transport can result in unbounded latency.
- The messages must pass through a restrictive firewall which does not pass multicast traffic.

Some users choose TCP to avoid unrecoverable loss. However, be aware that network failures can result in TCP disconnects, which can lead to missed messages. Also, if the DRO is being used, messages can be dropped due to full queues in an overloaded DRO, leading to unrecoverable loss.

UM's TCP transport includes a Session ID. A UM source using the TCP transport generates a unique, 32-bit non-zero random Session ID for each TCP transport (IP:port) it uses. The source also includes the Session ID in its Topic Resolution advertisement (TIR). When a receiver resolves its topic and discovers the transport information, the receiver also obtains the transport's Session ID. The receiver sends a message to the source to confirm the Session ID.

The TCP Session ID enables multiple receivers for a topic to connect to a source across a DRO. In the event of a DRO failure, UM establishes new topic routes which can cause cached Topic Resolution and transport information to be outdated. Receivers use this cached information to find sources. Session IDs add a unique identifier to the cached transport information. If a receiver tries to connect to a source with outdated transport information, the source recognizes an incorrect Session ID and disconnects the receiver. The receiver can then attempt to reconnect with different cached transport information.

Note

> To maintain interoperability between version pre-6.0 receivers and version 6.0 and beyond TCP sources, you can turn off TCP Session IDs with the UM configuration option, **transport_tcp_use_session_id (source)**.

6.1.1 TCP Flow Control Restrictions

The TCP transport protocol can provide a limited form of flow control. I.e. if a publisher is sending messages faster than a subscriber can process them, the publisher's "send" call will "block" to force the sender to slow down to the receiver's rate. (Alternatively, a non-blocking send will return -1 with the error code **LBM_EWOULDBLOCK**.)

However, be aware that there are restrictions on this use case.

- **DRO** - The DRO does not support end-to-end flow control. If a DRO is between a source and a slow receiver, the DRO will drop messages, resulting in unrecoverable loss in the receiver.
- **Event Queue** - If the receiver uses an event queue, a slow receiver can result in unbounded memory growth in the subscriber, rather than slowing down the publisher.
- **transport_tcp_multiple_receiver_behavior (source)** - if set to "source_paced", the source will drop messages intended for a slow receiver when the socket buffers fill, resulting in unrecoverable loss in the receiver.

If end-to-end application-level flow control is needed, users should implement their own handshakes.

6.2 Transport LBT-RU

The LBT-RU UM transport adds reliable delivery to unicast UDP to send messages from sources to receivers. This provides greater flexibility in the control of latency. For example, the application can further limit latency by allowing the use of arrival order delivery. See the Knowledge Base article, `FAQ: How do arrival-order delivery and in-order delivery affect latency?`. Also, LBT-RU is less sensitive to overall network load; it uses source rate controls to limit its maximum send rate.

LBT-RU's Transport Pacing is source-paced.

Since it is based on unicast addressing, LBT-RU can pass through most firewalls. However, it has the same scaling issues as TCP when multiple receivers are present for each topic.

UM's LBT-RU transport includes a Session ID. A UM source using the LBT-RU transport generates a unique, 32-bit non-zero random Session ID for each transport it uses. The source also includes the Session ID in its Topic Resolution advertisement (TIR). When a receiver resolves its topic and discovers the transport information, the receiver also obtains the transport's Session ID.

The LBT-RU Session ID enables multiple receivers for a topic to connect to a source across a DRO. In the event of a DRO failure, UM establishes new topic routes which can cause cached Topic Resolution and transport information to be outdated. Receivers use this cached information to find sources. Session IDs add a unique identifier to the cached transport information. If a receiver tries to connect to a source with outdated transport information, the transport drops the received data and times out. The receiver can then attempt to reconnect with different cached transport information.

Note

> To maintain interoperability between version pre-3.3 receivers and version 3.3 and beyond LBT-RU sources, you can turn off LBT-RU Session IDs with the UM configuration option, **transport_lbtru_use_session_id (source)**.
> LBT-RU can benefit from hardware acceleration. See **Transport Acceleration Options** for more information.

6.3 Transport LBT-RM

The LBT-RM transport adds reliable multicast to UDP to send messages. This provides the maximum flexibility in the control of latency. In addition, LBT-RM can scale effectively to large numbers of receivers per topic using network hardware to duplicate messages only when necessary at wire speed. One limitation is that multicast is often blocked by firewalls.

LBT-RM's Transport Pacing is source-paced.

LBT-RM is a UDP-based, reliable multicast protocol designed with the use of UM and its target applications specifically in mind.

UM's LBT-RM transport includes a Session ID. A UM source using the LBT-RM transport generates a unique, 32-bit non-zero random Session ID for each transport it uses. The source also includes the Session ID in its Topic Resolution advertisement (TIR). When a receiver resolves its topic and discovers the transport information, the receiver also obtains the transport's Session ID.

Note

> LBT-RM can benefit from hardware acceleration. See **Transport Acceleration Options** for more information.

6.3.1 NAK Suppression

Some reliable multicast protocols are susceptible to "NAK storms" in which a subscriber experiences loss due to overload, and the publisher sends those retransmissions, which makes the subscriber's overload *more severe*, resulting in an even greater loss rate, which triggers an even greater NAK rate, which triggers an even greater retransmission rate, and so on. This self-reinforcing feedback loop can cause healthy, non-overloaded subscribers to become so overwhelmed with retransmissions that they also experience loss and send NAKs, making the storm worse. Some users of other protocols have experienced NAK storms so severe that their entire network "melts down" and becomes unresponsive.

The Ultra Messaging LBT-RM protocol was designed specifically to prevent this kind of run-away feedback loop using a set of algorithms collectively known as "NAK Suppression". These algorithms are designed to repair occasional loss reasonably quickly, while preventing a crybaby receiver from degrading the overall system.

Note that although UM's Transport LBT-RU does not use Multicast, there are still dangers associated with crybaby receivers. LBT-RU implements most of the same NAK suppression algorithms.

Here are the major elements of UM's NAK suppression algorithms:

- When a receiver detects datagram loss (missing sequence number), it chooses a random amount of time to back off before sending a NAK (see **transport_lbtrm_nak_initial_backoff_interval (receiver)**). If the receiver subsequently receives the missing datagram(s) before the timer expires, it cancels the timer and sends no NAK.

  This reduces the load in the case of common loss patterns across multiple receivers. Whichever receiver chooses the smallest amount of time sends the NAK, and the retransmission prevents (suppresses) NAKs from the other receivers.

- The source has a configurable limit on the rate of retransmissions allowed; see **transport_lbtrm_↩ retransmit_rate_limit (context)**. When that rate is reached, the source rejects NAKs for the remainder of the rate limit interval and instead sends NCFs in response to the NAKs. (NCFs suppress receivers from sending NAKs for specific sequence numbers for configurable periods of time.)

- The source has a configurable interval during which it will only send one retransmission for a given sequence number; see **transport_lbtrm_ignore_interval (source)**. If a source receives multiple NAKs for the same sequence number within that interval, only the first will trigger a retransmission. Subsequent NAKs within the ignore interval are rejected, triggering one or more NCFs.

Note that LBT-RM's NCF algorithm was improved in UM version 6.10. Prior to 6.10, the source responds to every rejected NAK with an NCF (if the reason for the rejection is temporary). In UM 6.10 and beyond, only the first rejected NAK within an ignore interval triggers an NCF. Any subsequent NAKs in the same ignore interval are silently discarded. This reduces the NCF rate in a wide variety of loss conditions, which reduces the stress on healthy receivers.

Warning

> We have seen customers choose configuration values which reduce the protective characteristics of LBT-RM. For example, setting the retransmission rate too high, or the NAK backoff intervals too low. Doing so can reduce loss-induced latency, but can also risk disruptive NAK/retransmission storms which can stress otherwise healthy receivers. Users should request review of their configurations from Informatica's Ultra Messaging support organization.

6.3.2 Comparing LBT-RM and PGM

The LBT-RM protocol is very similar to PGM, but with changes to aid low latency messaging applications.

- Topic Mapping - Several topics may map onto the same LBT-RM session. Thus a multiplexing mechanism to LBT-RM is used to distinguish topic level concerns from LBT-RM session level concerns (such as retransmissions, etc.). Each message to a topic is given a sequence number in addition to the sequence number used at the LBT-RM session level for packet retransmission.

- Negative Acknowledgments (NAKs) - LBT-RM uses NAKs as PGM does. NAKs are unicast to the sender. For simplicity, LBT-RM uses a similar NAK state management approach as PGM specifies.

- Time Bounded Recovery - LBT-RM allows receivers to specify a maximum time to wait for a lost piece of data to be retransmitted. This allows a recovery time bound to be placed on data that has a definite lifetime of usefulness. If this time limit is exceeded and no retransmission has been seen, then the piece of data is marked as an unrecoverable loss and the application is informed. The data stream may continue and the unrecoverable loss will be ordered as a discrete event in the data stream just as a normal piece of data.

- Flexible Delivery Ordering - LBT-RM receivers have the option to have the data for an individual topic delivered "in order" or "arrival order". Messages delivered "in order" will arrive in sequence number order to the application. Thus loss may delay messages from being delivered until the loss is recovered or unrecoverable loss is determined. With "arrival-order" delivery, messages will arrive at the application as they are received by the LBT-RM session. Duplicates are ignored and lost messages will have the same recovery methods applied, but the ordering may not be preserved. Delivery order is a topic level concern. Thus loss of messages in one topic will not interfere or delay delivery of messages in another topic.

- Session State Advertisements - In PGM, SPM packets are used to advertise session state and to perform PGM router assist in the routers. For LBT-RM, these advertisements are only used when data are not flowing. Once data stops on a session, advertisements are sent with an exponential back-off (to a configurable maximum interval) so that the bandwidth taken up by the session is minimal.

- Sender Rate Control - LBT-RM can control a sender's rate of injection of data into the network by use of a rate limiter. This rate is configurable and will back pressure the sender, not allowing the application to exceed the rate limit it has specified. In addition, LBT-RM senders have control over the rate of retransmissions separately from new data. This allows the publisher to guarantee a minimum transmission rate even in the face of massive loss at some or all receivers.

- Low Latency Retransmissions - LBT-RM senders do not mandate the use of NCF packets as PGM does. Because low latency retransmissions is such an important feature, LBT-RM senders by default send retransmissions immediately upon receiving a NAK. After sending a retransmission, the sender ignores additional NAKs for the same data and does not repeatedly send NCFs. The oldest data being requested in NAKs has priority over newer data so that if retransmissions are rate controlled, then LBT-RM sends the most important retransmissions as fast as possible.

6.4 Transport LBT-IPC

The LBT-IPC transport is an Interprocess Communication (IPC) UM transport that allows sources to publish topic messages to a shared memory area managed as a static ring buffer from which receivers can read topic messages. Message exchange takes place at memory access speed which can greatly improve throughput when sources and receivers can reside on the same host.

LBT-IPC's Transport Pacing can be either source-paced or receiver-paced. See **transport_lbtipc_behavior (source)**.

The LBT-IPC transport uses a "lock free" design that eliminates calls to the Operating System and allows receivers quicker access to messages. An internal validation method enacted by receivers while reading messages from the Shared Memory Area ensures message data integrity. The validation method compares IPC header information at different times to ensure consistent, and therefore, valid message data. Sources can send individual messages or a batch of messages, each of which possesses an IPC header.

Note that while the use of Transport Services Provider (XSP) for network-based transports does not conflict with LBT-IPC, it does not apply to LBT-IPC. I.e. IPC data flows cannot be assigned to XSP threads.

6.4.1 Sources and LBT-IPC

When you create a source with **lbm_src_create()** and you've set the transport option to IPC, UM creates a shared memory area object. UM assigns one of the transport IDs to this area specified with the UM context configuration options, **transport_lbtipc_id_high (context)** and **transport_lbtipc_id_low (context)**. You can also specify a shared memory location outside of this range with a source configuration option, **transport_lbtipc_id (source)**, to prioritize certain topics, if needed.

UM names the shared memory area object according to the format, LBTIPC_x_d where x is the hexadecimal Session ID and d is the decimal Transport ID. Example names are **LBTIPC_42792ac_20000** or **LBTIPC_66e7c8f6↩ _20001**. Receivers access a shared memory area with this object name to receive (read) topic messages.

See **Transport LBT-IPC Operation Options** for configuration information.

Sending over LBT-IPC

To send on a topic (write to the shared memory area) the source writes to the Shared Memory Area starting at the Oldest Message Start position. It then increments each receiver's Signal Lock if the receiver has not set this to zero.

6.4.2 Receivers and LBT-IPC

Receivers operate identically to receivers for all other UM transports. A receiver can actually receive topic messages from a source sending on its topic over TCP, LBT-RU or LBT-RM and from a second source sending on LBT-IPC with out any special configuration. The receiver learns what it needs to join the LBT-IPC session through the topic advertisement.

The configuration option **transport_lbtipc_receiver_thread_behavior (context)** controls the IPC receiving thread behavior when there are no messages available. The default behavior, '**pend**', has the receiving thread pend on a semaphore for a new message. When the source adds a message, it posts to each pending receiver's semaphore to wake the receiving thread up. Alternatively, busy_wait can be used to prevent the receiving thread going to sleep. In this case, the source does not need to post to the receiver's semaphore. It simply adds the message to shared memory, which the looping receiving thread detects with the lowest possible latency.

Although '**busy_wait**' has the lowest latency, it has the drawback of consuming 100% of a CPU core during periods of idleness. This limits the number of IPC data flows that can be used on a given machine to the number of available cores. (If more busy looping receivers are deployed than there are cores, then receivers can suffer 10 millisecond time sharing quantum latencies.)

For application that cannot afford '**busy_wait**', there is another configuration option, **transport_lbtipc_pend_behavior_linger_loop_count (context)**, which allows a middle ground between '**pend**' and '**busy_wait**'. The receiver is still be configured as '**pend**', but instead of going to sleep on the semaphore *immediately* upon emptying the shared memory, it busy waits for the configured number of times. If a new message arrives, it processes the message immediately without a sleep/wakeup. This can be very useful during bursts of high incoming message rates to reduce latency. By making the loop count large enough to cover the incoming message interval during a burst, only the first message of the burst will incur the wakeup latency.

Topic Resolution and LBT-IPC

Topic resolution operates identically with LBT-IPC as other UM transports albeit with a new advertisement type, LBMIPC. Advertisements for LBT-IPC contain the Transport ID, Session ID and Host ID. Receivers obtain LBT-IPC advertisements in the normal manner (resolver cache, advertisements received on the multicast resolver address:port and responses to queries.) Advertisements for topics from LBT-IPC sources can reach receivers on different machines if they use the same topic resolution configuration, however, those receivers silently ignore those advertisements since they cannot join the IPC transport. See Sending to Both Local and Remote Receivers.

Receiver Pacing

Although receiver pacing is a source behavior option, some different things must happen on the receiving side to ensure that a source does not reclaim (overwrite) a message until all receivers have read it. When you use the default **transport_lbtipc_behavior (source)** (source-paced), each receiver's Oldest Message Start position in the Shared Memory Area is private to each receiver. The source writes to the Shared Memory Area independently of receivers' reading. For receiver-pacing, however, all receivers share their Oldest Message Start position with the source. The source will not reclaim a message until all receivers have successfully read that message.

Receiver Monitoring

To ensure that a source does not wait on a receiver that is not running, the source monitors a receiver via the Monitor Shared Lock allocated to each receiving context. (This lock is in addition to the semaphore already allocated for signaling new data.) A new receiver takes and holds the Monitor Shared Lock and releases the resource when it dies. If the source is able to obtain the resource, it knows the receiver has died. The source then clears the receiver's In Use flag in it's Receiver Pool Connection.

6.4.3 Similarities with Other UM Transports

Although no actual network transport occurs, IPC functions in much the same way as if you send packets across the network as with other UM transports.

- If you use a range of LBT-IPC transport IDs, UM assigns multiple topics sent by multiple sources to all the Transport Sessions in a round robin manner just like other UM transports.

- Transport sessions assume the configuration option values of the first source assigned to the Transport Session.

- Sources are subject to message batching.

6.4.4 Differences from Other UM Transports

- Unlike LBT-RM which uses a transmission window to specify a buffer size to retain messages in case they must be retransmitted, LBT-IPC uses the transmission window option to establish the size of the shared memory.

- LBT-IPC does not retransmit messages. Since LBT-IPC transport is essentially a memory write/read operation, messages should not be be lost in transit. However, if the shared memory area fills up, new messages overwrite old messages and the loss is absolute. No retransmission of old messages that have been overwritten occurs. (Note: while transport-level retransmission is not available, IPC is compatible with the Off-Transport Recovery (OTR) feature, which allows for persistent message recovery from the Store, or streaming message recovery from the Source's Late Join buffer.)

- Receivers also do not send NAKs when using LBT-IPC.

- LBT-IPC does not support **ordered_delivery (receiver)** options. However, if you set **ordered_delivery (receiver)** 1 or -1, LBT-IPC reassembles any large messages.

- LBT-IPC does not support Rate Limiting.

- LBT-IPC creates a separate receiver thread in the receiving context.

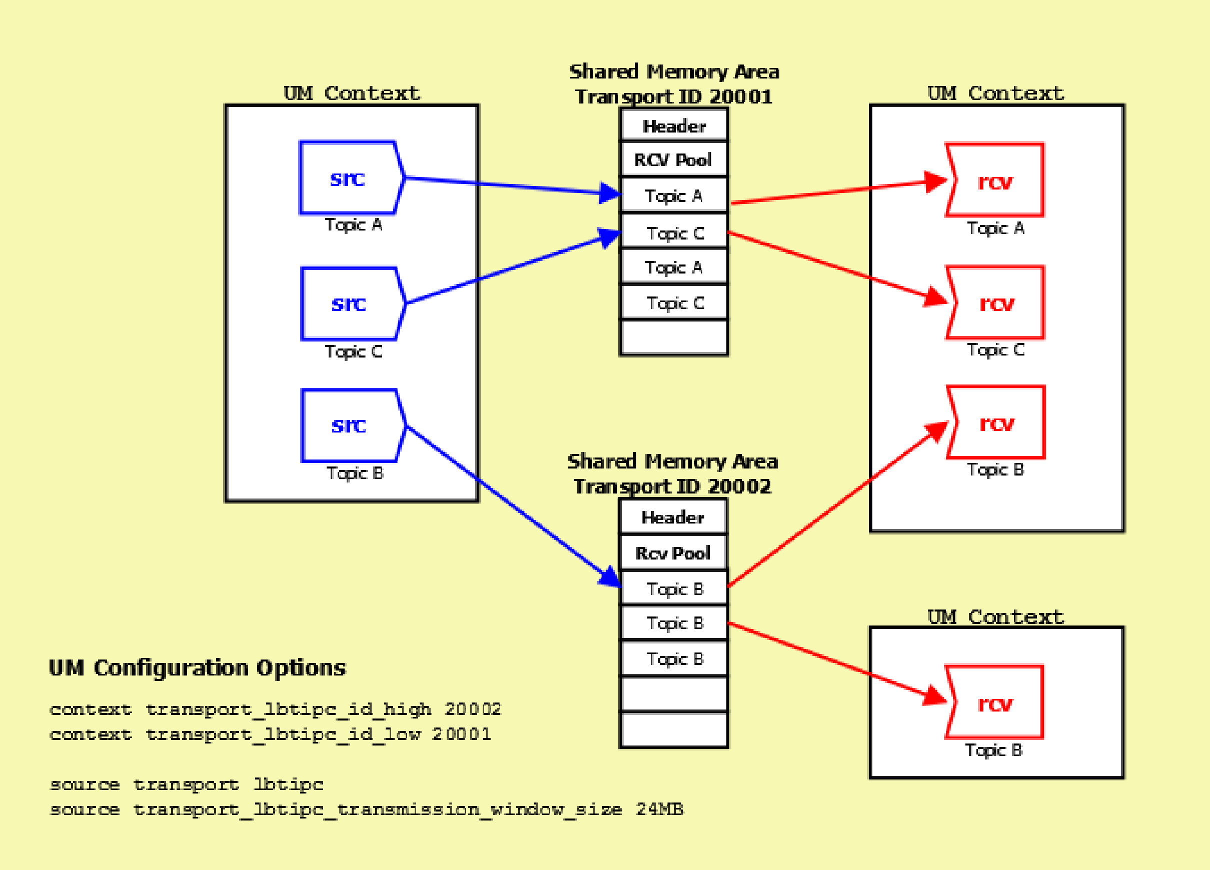
6.4.5 Sending to Both Local and Remote Receivers

A source application that wants to support both local and remote receivers should create two UM Contexts with different topic resolution configurations, one for IPC sends and one for sends to remote receivers. Separate contexts allows you to use the same topic for both IPC and network sources. If you simply created two source objects (one IPC, one say LBT-RM) in the same UM Context, you would have to use separate topics and suffer possible higher latency because the sending thread would be blocked for the duration of two send calls.

A UM source will never automatically use IPC when the receivers are local and a network transport for remote receivers because the discovery of a remote receiver would hurt the performance of local receivers. An application that wants transparent switching can implement it in a simple wrapper.

6.4.6 LBT-IPC Configuration Example

The following diagram illustrates how sources and receivers interact with the shared memory area used in the LBT-IPC transport:



UM Configuration Options

```
context transport_lbtipc_id_high 20002
context transport_lbtipc_id_low 20001

source transport lbtipc
source transport_lbtipc_transmission_window_size 24MB
```

In the diagram above, 3 sources send (write) to two Shared Memory Areas while four receivers in two different contexts receive (read) from the areas. The assignment of sources to Shared Memory Areas demonstrate UM's round robin method. UM assigns the source sending on Topic A to Transport 20001, the source sending on Topic B to Transport 20002 and the source sending on Topic C back to the top of the transport ID range, 20001.

The diagram also shows the UM configuration options that set up this scenario:

- The options **transport_lbtipc_id_high (context)** and **transport_lbtipc_id_low (context)** establish the range of Transport IDs between 20001 and 20002.
- The option **transport (source)** is used to set the source's transport to LBT-IPC.
- The option **transport_lbtipc_transmission_window_size (source)** sets the size of each Shared Memory Area to 24 MB.

6.4.7 Required privileges

LBT-IPC requires no special operating system authorities, except on Microsoft Windows Vista and Microsoft Windows Server 2008, which require Administrator privileges. In addition, on Microsoft Windows XP, applications must be started by the same user, however, the user is not required to have administrator privileges. In order for applications to communicate with a service, the service must use a user account that has Administrator privileges.

6.4.8 Host Resource Usage and Limits

LBT-IPC contexts and sources consume host resources as follows:

- Per Source - 1 shared memory area, 1 shared lock (semaphore on Linux, mutex on Microsoft Windows)
- Per Receiving Context - 2 shared locks (semaphores on Linux, one event and one mutex on Microsoft Windows)

Across most operating system platforms, these resources have the following limits.

- 4096 shared memory areas, though some platforms use different limits
- 32,000 shared semaphores (128 shared semaphore sets $*$ 250 semaphores per set)

Consult your operating system documentation for specific limits per type of resource. Resources may be displayed and reclaimed using the LBT-IPC Resource Manager. See also the KB article `Managing LBT-IPC Host Resources`.

6.4.9 LBT-IPC Resource Manager

Deleting an IPC source or deleting an IPC receiver reclaims the shared memory area and locks allocated by the IPC source or receiver. However, if a less than graceful exit from a process occurs, global resources remain allocated but unused. To address this possibility, the LBT-IPC Resource Manager maintains a resource allocation database with a record for each global resource (memory or semaphore) allocated or freed. You can use the LBT-IPC Resource Manager to discover and reclaim resources. See the three example outputs below.

Displaying Resources

```
$> lbtipc_resource_manager
Displaying Resources (to reclaim you must type '-reclaim' exactly)

--Memory Resources--
Memory resource: Process ID: 24441 SessionID: ab569cec XportID: 20001

--Semaphore Resources-- Semaphore key: 0x68871d75
Semaphore resource Index 0: reserved

Semaphore resource: Process ID: 24441 Sem Index: 1
Semaphore resource: Process ID: 24436 Sem Index: 2
```

Reclaiming Unused Resources

```
$> lbtipc_resource_manager -reclaim
Reclaiming Resources
Process 24441 not found: reclaiming Memory resource (SessionID: ab569cec XPortID:
    20001)
Process 24441 not found: reclaiming Semaphore resource: Key: 0x68871d75 Sem Index: 1
Process 24436 not found: reclaiming Semaphore resource: Key: 0x68871d75 Sem Index: 2
```

6.5 Transport LBT-SMX

The LBT-SMX (shared memory acceleration) transport is an Interprocess Communication (IPC) transport you can use for the lowest latency message Streaming.

LBT-RU's Transport Pacing is receiver-paced. If you need source pacing, see Transport LBT-IPC.

Like LBT-IPC, sources can publish topic messages to a shared memory area from which receivers can read topic messages. LBT-SMX is slightly faster than Transport LBT-IPC. However, SMX imposes more limitations than LBT-IPC; see Differences Between LBT-SMX and Other UM Transports.

To achieve the minimum possible latency and jitter, the receive side uses busy waiting exclusively. This means that while waiting for a message, the receiving context has a thread running at 100% CPU utilization. To avoid latency jitter, Informatica strongly recommends pinning the thread to a single core to prevent the operating system from migrating the thread across cores.

Although busy waiting has the lowest latency, it has the drawback of consuming 100% of a CPU core, even during periods of no messages. This limits the number of SMX data flows that can be used on a given machine to the number of available cores. If this is not practical for your use case, Informatica recommends the use of Transport LBT-IPC.

There are two forms of API calls available with LBT-SMX:

- General APIs - the same APIs used with other transports, such as sending messages with **lbm_src_send()**.
- SMX-specific APIs - APIs that further reduce overhead and latency but require a somewhat different code design, such as sending messages with **lbm_src_buff_acquire()** and **lbm_src_buffs_complete()**.

Note that the SMX-specific APIs are not thread safe, and require that the calling application guarantee serialization of calls on a given Transport Sessions.

LBT-SMX operates on the following Ultra Messaging platforms:

- 64-bit SunOS (X86 only)
- 64-bit Linux
- 64-bit Windows

The "latency ping/pong" example programs demonstrate how to use the SMX-specific API.

For C: **Example lbmlatping.c** and **Example lbmlatpong.c**.

For Java: **Example lbmlatping.java** and **Example lbmlatpong.java**.

For .NET: **Example lbmlatping.cs** and **Example lbmlatpong.cs**.

Many other example applications can use the LBT-SMX transport by setting the **transport (source)** configuration option to "LBT-SMX":

```
source transport LBT-SMX
```

However, you cannot use LBT-SMX with example applications for features not supported by LBT-SMX, such as **Example lbmreq.c**, **Example lbmresp.c**, **Example lbmrcvq.c**, or **Example lbmwrcvq.c**.

The LBT-SMX configuration options are similar to the LBT-IPC transport options. See **Transport LBT-SMX Operation Options** for more information.

You can use Automatic Monitoring, UM API retrieve/reset calls, and LBMMON APIs to access LBT-SMX source and receiver transport statistics. To increase performance, the LBT-SMX transport does not collect statistics by default. Set the UM configuration option **transport_lbtsmx_message_statistics_enabled (context)** to 1 to enable the collection of transport statistics.

The next few sections provide an in-depth understanding of how SMX works, using the C API as the model. Java and .NET programmers are encouraged to read these sections for the understanding of SMX. There are also sections specific to Java and .NET near the end to provide API details.

6.5.1 Sources and LBT-SMX

When you create a source with **lbm_src_create()** and you've set the **transport (source)** option to "LBT-SMX", UM creates a shared memory area object. UM assigns one of the transport IDs to this area from a range of transport IDs specified with the UM context configuration options **transport_lbtsmx_id_high (context)** and **transport_lbtsmx_id_low (context)**. You can also specify a shared memory location inside or outside of this range with the option **transport_lbtsmx_id (source)** to group certain topics in the same shared memory area, if needed. See **Transport LBT-SMX Operation Options** for configuration details.

UM names the shared memory area object according to the format, **LBTSMX_x_d** where **x** is the hexadecimal Session ID and **d** is the decimal Transport ID. Example names are **LBTSMX_42792ac_20000** or **LBTSMX_66e7c8f6_20001**. Receivers access a shared memory area with this object name to receive (read) topic messages.

Note

> For every context created by your application, UM creates an additional shared memory area for control information. The name for these control information memory areas ends with the suffix, _0, which is the Transport ID.

Here are the SMX-specific in C:

- **lbm_src_buff_acquire()** - obtains a pointer into the source's shared buffer for a new message of the specified length.
- **lbm_src_buffs_complete()** - informs UM that one or more acquired messages are complete and ready to be delivered to receivers.
- **lbm_src_buffs_complete_and_acquire()** - a convenience function that is the same as calling **lbm_src_buff_acquire()** followed immediately by **lbm_src_buffs_complete()**.
- **lbm_src_buffs_cancel()** - cancels all message buffers acquired but not yet completed.

The SMX-specific APIs fail with an appropriate error message if a sending application uses them for a source configured to use a transport other than LBT-SMX.

Note

> The SMX-specific APIs are not thread safe at the source object or LBT-SMX Transport Session levels for performance reasons. Applications that use the SMX-specific APIs for either the same source or a group of sources that map to the same LBT-SMX Transport Session must serialize the calls either directly in the application or through their own mutex.

6.5.2 Sending with SMX-specific APIs

Sending with SMX-specific APIs is a two-step process.

1. The publisher first calls **lbm_src_buff_acquire()**, which returns a pointer into which the publisher writes the message data.

   The pointer points directly into the shared memory area. UM guarantees that the shared memory area has at least the value specified with the len parameter of contiguous bytes available for writing when **lbm_src_buff_acquire()** returns. Note that the acquire function has the potential of blocking (spinning), if the shared memory area is full of messages that are unread by at least one subscribed receiver. If your application set the LBM_SRC_NONBLOCK flag with **lbm_src_buff_acquire()**, UM immediately returns an LBM_EWOULDBLOCK error condition if the function detects the blocking condition.

   Because LBT-SMX does not support fragmentation, your application must limit message lengths to a maximum equal to the value of the source's configured **transport_lbtsmx_datagram_max_size (source)** option minus 16 bytes for headers. In a system deployment that includes the DRO, this value should be the same as the Datagram Max Sizes of other transport types. See **Protocol Conversion**.

After the user acquires the pointer into shared memory and writes the message data, the application may call **lbm_src_buff_acquire()** repeatedly to send a batch of messages to the shared memory area. If your application writes multiple messages in this manner, sufficient space must exist in the shared memory area. **lbm_src_buff_acquire()** returns an error if the available shared memory space is less than the size of the next message.

2. The publisher calls one of the two following APIs.

   - **lbm_src_buffs_complete()**, which publishes the message or messages to all listening receivers.
   - **lbm_src_buffs_complete_and_acquire()**, which publishes the message or messages to all listening receivers and acquires a new pointer.

6.5.3 Sending over LBT-SMX with General APIs

LBT-SMX supports the general send API functions, like **lbm_src_send()**. These API calls are fully thread-safe. The LBT-SMX feature restrictions still apply (see Differences Between LBT-SMX and Other UM Transports). The lbm_src_send_ex_info_t argument to the **lbm_src_send_ex()** and **lbm_src_sendv_ex()** APIs must be NULL when using an LBT-SMX source, because LBT-SMX does not support any of the features that the lbm_src_send_ex_↩ info_t parameter can enable.

Since LBT-SMX does not support an implicit batcher or corresponding implicit batch timer, UM flushes all messages for all sends on LBT-SMX transports done with general APIs, which is similar to setting the LBM_MSG_FLUSH flag. LBT-SMX also supports the **lbm_src_flush()** API call, which behaves like a thread-safe version of **lbm_src_buffs↩ _complete()**.

Warning

> Users should not use both the SMX-specific APIs and the general API calls in the same application. Users should choose one or the other type of API for consistency and to avoid thread safety problems.

The **lbm_src_topic_alloc()** API call generates log warnings if the given attributes specify an LBT-SMX transport and enable any of the features that LBT-SMX does not support. The **lbm_src_topic_alloc()** returns 0 (success), but UM does not enable the unsupported features indicated in the log warnings. Other API functions that operate on lbm_src_t objects, such as **lbm_src_create()**, **lbm_src_delete()**, or **lbm_src_topic_dump()**, operate with LBT-SMX sources normally.

Because LBT-SMX does not support fragmentation, your application must limit message lengths to a maximum equal to the value of the source's configured **transport_lbtsmx_datagram_max_size (source)** option minus 16 bytes for headers. Any send API calls with a length parameter greater than this configured value fail. In a system deployment that includes the DRO, this value should be the same as the Datagram Max Sizes of other transport types. See **Protocol Conversion**.

6.5.4 Receivers and LBT-SMX

LBT-SMX receivers can be coded the same as receivers on other UM transports. The msg->data pointer of a delivered lbm_msg_t object points directly into the shared memory area. However, Java and .NET receivers can benefit from some alternate coding techniques. See Java Coding for LBT-SMX and .NET Coding for LBT-SMX.

The **lbm_msg_retain()** API function operates differently for LBT-SMX. **lbm_msg_retain()** creates a full copy of the message in order to access the data outside the receiver callback.

Attention

> You application should not pass the msg->data pointer to other threads or outside the receiver callback until your application has called **lbm_msg_retain()** on the message.

Warning

> Any API calls documented as not safe to call from a context thread callback are also not safe to call from an LBT-SMX receiver thread.

Topic Resolution and LBT-SMX

Topic resolution operates identically with LBT-SMX as other UM transports albeit with the advertisement type, LBMSMX. Advertisements for LBT-SMX contain the Transport ID, Session ID, and Host ID. Receivers get LBT-SMX advertisements in the normal manner, either from the resolver cache, advertisements received on the multicast resolver address:port, or responses to queries.

6.5.5 Similarities Between LBT-SMX and Other UM Transports

Although no actual network transport occurs, SMX functions in much the same way as if you send packets across the network as with other UM transports.

- If you use a range of LBT-SMX transport IDs, UM assigns multiple topics sent by multiple sources to all the Transport Sessions in a round robin manner just like other UM transports.
- Transport sessions assume the configuration option values of the first source assigned to the Transport Session.
- Source applications and receiver applications based on any of the three available APIs can interoperate with each other. For example, sources created by a C publisher can send to receivers created by a Java receiving application.

6.5.6 Differences Between LBT-SMX and Other UM Transports

- Unlike LBT-RM which uses a transmission window to specify a buffer size to retain messages for retransmission, LBT-SMX uses the transmission window option to establish the size of the shared memory. LBT-SMX uses transmission window sizes that are powers of 2. You can set **transport_lbtsmx_transmission_window_size (source)** to any value, but UM rounds the option value up to the nearest power of 2.
- The largest transmission window size for Java applications is 1 GB.
- The largest possible message size for Java applications is 1 GB.
- LBT-SMX does not retransmit messages. Since LBT-SMX transport only supports receiver pacing, messages are never lost in transit.
- Receivers do not send NAKs when using LBT-SMX.

LBT-SMX is not compatible with the following UM features:

- Arrival Order, Fragments Not Reassembled (ordered_delivery 0).
- Source Pacing.

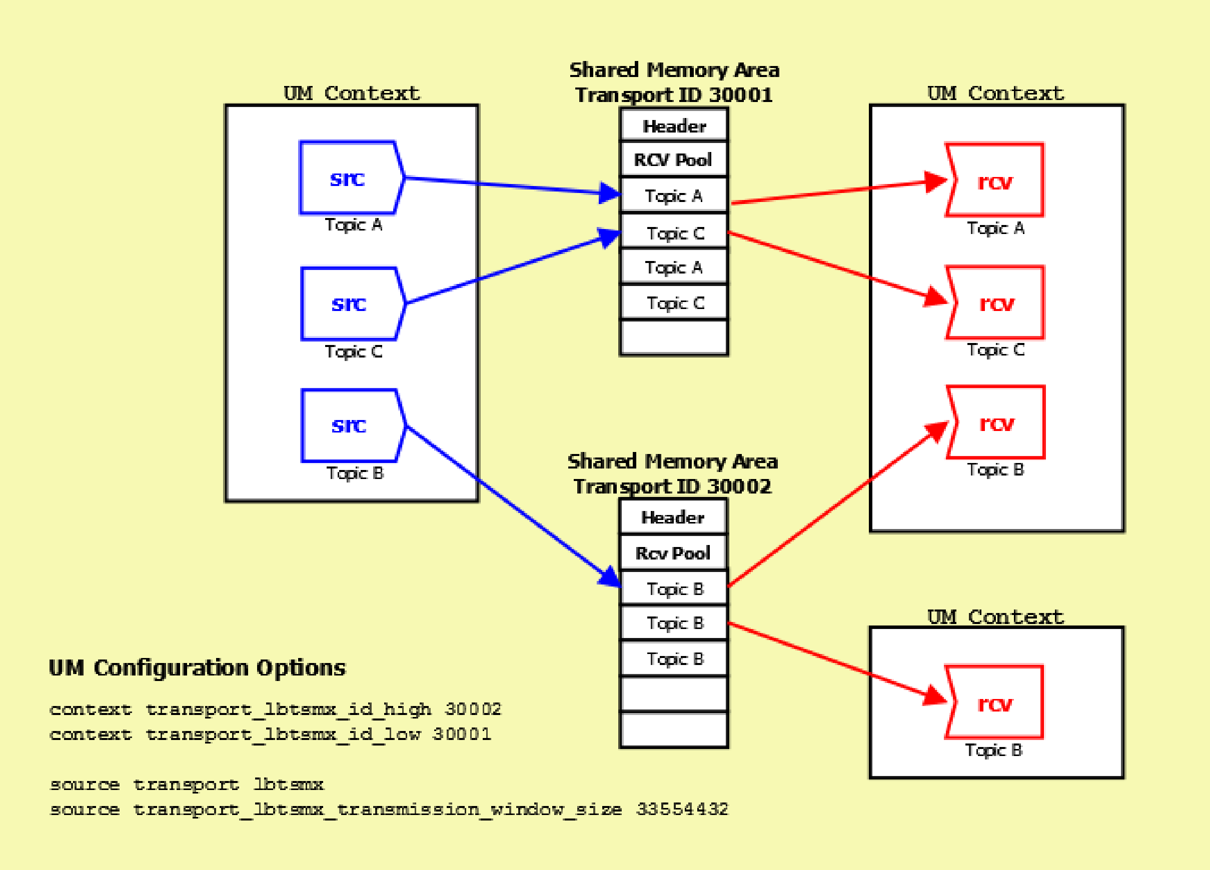
- Late Join.
- Off-Transport Recovery (OTR).
- Request/Response.
- **Source-side Filtering**.
- Hot Failover (HF).
- Message Properties.
- Application Headers.
- Implicit Batching and Explicit Batching.
- Message Fragmentation and Reassembly.
- Unicast Immediate Messaging.
- Multicast Immediate Messaging.
- The "pend"-style of Receiver thread behavior; SMX only supports busy_wait-style.
- Persistence.
- Queuing.

Note that while the use of Transport Services Provider (XSP) for network-based transports does not conflict with LBT-SMX, it does not apply to LBT-SMX. I.e. SMX data flows cannot be assigned to XSP threads.

You also cannot use LBT-SMX to send outgoing traffic from a UM daemon, such as the persistent Store, DRO, or UMDS.

6.5.7 LBT-SMX Configuration Example

The following diagram illustrates how sources and receivers interact with the shared memory area used in the LBT-SMX transport.



```
context transport_lbtsmx_id_high 30002
context transport_lbtsmx_id_low 30001

source transport lbtsmx
source transport_lbtsmx_transmission_window_size 33554432
```

In the diagram above, three sources send (write) to two Shared Memory Areas while four receivers in two different contexts receive (read) from the areas. The assignment of sources to Shared Memory Areas demonstrate UM's round robin method. UM assigns the source sending on Topic A to Transport 30001, the source sending on Topic B to Transport 30002 and the source sending on Topic C back to the top of the transport ID range, 30001.

The diagram also shows the UM configuration options that set up this scenario.

- The options **transport_lbtsmx_id_high (context)** and **transport_lbtsmx_id_low (context)** establish the range of Transport IDs between 30001 and 30002.
- The option "source transport lbtsmx" sets the source's transport to LBT-SMX.
- The option **transport_lbtsmx_transmission_window_size (source)** sets the size of each Shared Memory Area to 33554432 bytes or 32 MB. This option's value must be a power of 2. If you configured the transmission window size to 25165824 bytes (24 MB) for example, UM logs a warning message and then rounds the value of this option up to the next power of 2 or 33554432 bytes or 32 MB.

6.5.8 Java Coding for LBT-SMX

Java-based SMX works by wrapping the shared memory area in a `ByteBuffer` object. This lets the sender call the put methods to build your application message directly in the shared memory, and the receiver calls the get methods to extract the message directly from the shared memory.

Here are the SMX-specific APIs in Java:

- **LBMSource.getMessagesBuffer()** - obtains a reference to the source's ByteBuffer, which does not change for the lifetime of the source object.

- **LBMSource.acquireMessageBufferPosition()** - obtains an offset into the source's ByteBuffer for a new message of the specified length.
- **LBMSource.messageBuffersComplete()** - informs UM that one or more acquired messages are complete and ready to be delivered to receivers.
- **LBMSource.messageBuffersCompleteAndAcquirePosition()** - a convenience function that is the same as calling messageBuffersComplete() followed immediately by acquireMessageBufferPosition().
- **LBMSource.messageBuffersCancel()** - cancels all message buffers acquired but not yet completed.

Notice that while the normal source send can accept either a byte array or a ByteBuffer, the Java SMX-specific APIs only accept a ByteBuffer. Also note that the ByteBuffer does not necessarily start at position 0. You must call acquireMessageBufferPosition() to determine the offset.

The Java example programs **Example lbmlatping.java** and **Example lbmlatpong.java** illustrates how a Java-based SMX source and receiver is written.

For message reception, see the "onReceive" function. Note that there is no SMX-specific code. However, to maintain high performance, the user should call **LBMSource.getMessagesBuffer()** to obtain direct access to the source's ByteBuffer. The position and limit set according to the message to be read.

To save on some front-end latency, the "lat" ping/pong programs pre-acquire a buffer which is filled in when a messages needs to be sent (the call to **src.acquireMessageBufferPosition()**). After each send, another buffer is acquired for the next (future) message (the call to **src.messageBuffersCompleteAndAcquirePosition()**). This moves the overhead of acquiring a message buffer to *after* the message was sent.

Batching

SMX does not support the normal UM feature Implicit Batching. However, it does support a method of Intelligent Batching whereby multiple messages may be added to the source's ByteBuffer before being released to UM for delivery.

Note that unlike network-based transports, batching will give only minor benefits for SMX, mostly in the form of reduced memory contention.

```
/* Acquire a position in the buffer */
buf.position(src.acquireMessageBufferPosition(msglen, 0));
... /* Construct a message in "buf". */

buf.position(src.acquireMessageBufferPosition(msglen, 0));
... /* Construct a second message in "buf". */

/* Tell UM that all acquired message buffers are now complete. */
src.messageBuffersComplete();
```

Notice that the same ByteBuffer "buf" is used for both messages. This is because it is backed by the entire shared memory buffer. The current position of "buf" is set to the start of the message space acquired by acquireMessage↩BufferPosition().

Non-blocking

In the example code presented, calls to acquireMessageBufferPosition() passed zero for flags. This means that the call has the potential to block. Blocking happens if the sender outpaces the slowest receiver and the shared buffer fills. Because of receiver pacing, the source must not overwrite sent messages that have not been consumed by one or more subscribed receivers.

It is also possible to pass LBM.SRC_NONBLOCK as a flag to acquireMessageBufferPosition(). In that case, acquireMessageBufferPosition() will immediately return -1 if the shared buffer does not have enough space for the request.

```
pos = src.acquireMessageBufferPosition(msgLength, flags);
if (pos == -1) {
  /* Can't send the message yet. Handle this error. */
}
else {
  /* Successful acquisition, set up buffer. */
  buf.position(pos);
  ... /* Construct a message in "buf". */
  /* Tell UM that all acquired message buffers are now complete. */
  src.messageBuffersComplete();
}
```

For performance reasons, acquireMessageBufferPosition() does not throw LBMEWouldBlock exceptions, like the standard send APIs do.

Reduce Overhead

Normally, UM allows an application to add multiple receiver callbacks for the same topic receiver. This adds a small amount of overhead for a feature that is not used very often. This overhead can be eliminated by setting the "single receiver callback" attribute when creating the receiver object. This is done by calling `LBMReceiverAttributes.enableSingleReceiverCallback()`.

```
LBMReceiverAttributes rattr = new LBMReceiverAttributes();
rattr.enableSingleReceiverCallback(true);
LBMTopic topic = ctx.lookupTopic("lbmpong/pong", rattr);

rcv = new LBMReceiver(ctx, topic, new LBMReceiverCallback(onReceive), null);
```

6.5.9 .NET Coding for LBT-SMX

For most of UM functionality, the Java and .NET APIs are used almost identically. However, given SMX's latency requirements, we took advantage of some .NET capabilities that don't exist in Java, resulting in a different set of functions. In particular, instead of wrapping the shared memory area in a ByteBuffer, the .NET SMX API uses a simple IntPtr to provide a pointer directly into the shared memory area.

Here are the SMX-specific APIs in .NET:

- `LBMSource.buffAcquire()` - obtains a pointer into the source's shared buffer for a new message of the specified length.
- `LBMSource.buffsComplete()` - informs UM that one or more acquired messages are complete and ready to be delivered to receivers.
- `LBMSource.buffsCompleteAndAcquire()` - a convenience function that is the same as calling buffsComplete() followed immediately by acquireMessageBufferPosition().
- `LBMSource.buffsCancel()` - cancels all message buffers acquired but not yet completed.
- `LBMMessage.dataPointerSafe()` - Returns an IntPtr into the shared memory at the start of the message.

Notice that while the normal source send can accept either a byte array or a ByteBuffer, the .NET SMX-specific APIs only work with an IntPtr. This is because the application builds the message directly in the shared memory.

The .NET example programs **Example lbmlatping.cs** and **Example lbmlatpong.cs** illustrates how a .NET-based SMX source and receiver is written.

For message reception, see the "onReceive" function in **Example lbmpong.cs**. Note that there doesn't need to be any SMX-specific code. However, to maintain high performance, the user should call `LBMMessage.dataPointerSafe()` - to obtain direct access to the source's shared memory.

To save on some front-end latency, the "lat" ping/pong programs pre-acquire a buffer which is filled in when a messages needs to be sent (the call to source.buffAcquire()). After each send, another buffer is acquired for the next (future) message (the call to source.buffsCompleteAndAcquire()). This moves the overhead of acquiring a message buffer to *after* the message was sent.

Batching

SMX does not support the normal UM feature Implicit Batching. However, it does support a method of Intelligent Batching whereby multiple messages may be added to the source's shared memory before being released to UM for delivery.

Note that unlike network-based transports, batching will give only minor benefits for SMX, mostly in the form of reduced memory contention.

```
/* bufferAcquired was already set up earlier. */
... /* Construct first message at "bufferAcquired". */

source.buffAcquire(out bufferAcquired, (uint)buffer.Length, 0);
... /* Construct second message at "bufferAcquired". */

/* Send both messages, and set up "bufferAcquired" for next message. */
source.buffsCompleteAndAcquire(out bufferAcquired, (uint)buffer.Length, 0);

/* Tell UM that all acquired message buffers are now complete. */
src.buffsComplete();
```

Non-blocking

In the example code presented, calls to buffAcquire() and buffsCompleteAndAcquire() are passed zero for flags. This means that the call has the potential to block. Blocking happens if the sender outpaces the slowest receiver and the shared buffer fills. Because of receiver pacing, the source must not overwrite sent messages that have not been consumed by one or more subscribed receivers.

It is also possible to pass LBM.SRC_NONBLOCK as a flag to buffAcquire(). In that case, buffAcquire() will immediately return -1 if the shared buffer does not have enough space for the request.

```
err = src.buffAcquire(out writeBuff, (uint)msglen, LBM.SRC_NONBLOCK);
if (err == -1) {
  /* Can't send the message yet. Handle this error. */
}
else {
  /* Successful acquisition, set up buffer. */
  ... /* Construct a message in "writeBuff". */
  /* Tell UM that all acquired message buffers are now complete. */
  src.buffsComplete();
}
```

For performance reasons, buffsComplete() does not throw LBMEWouldBlock exceptions, like the standard send APIs do.

Reduce Overhead

Normally, UM allows an application to add multiple receiver callbacks for the same topic receiver. This adds a small amount of overhead for a feature that is not used very often. This overhead can be eliminated by setting the "single receiver callback" attribute when creating the receiver object. This is done by calling `LBMReceiverAttributes.enableSingleReceiverCallback()`.

```
LBMReceiverAttributes rattr = new LBMReceiverAttributes();
rattr.enableSingleReceiverCallback(true);
LBMTopic topic = ctx.lookupTopic("lbmpong/pong", rattr);

rcv = new LBMReceiver(ctx, topic, new LBMReceiverCallback(onReceive), null);
```

6.5.10 LBT-SMX Resource Manager

Deleting an SMX source or deleting an SMX receiver reclaims the shared memory area and locks allocated by the SMX source or receiver. However, if an ungraceful exit from a process occurs, global resources remain allocated but unused. To address this possibility, the LBT-SMX Resource Manager maintains a resource allocation database with a record for each global resource (memory or semaphore) allocated or freed. You can use the LBT-SMX Resource Manager to discover and reclaim resources. See the three example outputs below.

Displaying Resources

```
$> lbtsmx_resource_manager
Displaying Resources (to reclaim you must type '-reclaim' exactly)
```

```
--Memory Resources--
Memory resource: Process ID: 24441 SessionID: ab569cec XportID: 20001

--Semaphore Resources-- Semaphore key: 0x68871d75
Semaphore resource Index 0: reserved

Semaphore resource: Process ID: 24441 Sem Index: 1
Semaphore resource: Process ID: 24436 Sem Index: 2
```

Reclaiming Unused Resources

Warning

> This operation should never be done while SMX-enabled applications or daemons are running. If you have lost or unused resources that need to be reclaimed, you should exit all SMX applications prior to running this command.

```
$> lbtsmx_resource_manager -reclaim
Reclaiming Resources
Process 24441 not found: reclaiming Memory resource (SessionID: ab569cec XPortID:
    20001)
Process 24441 not found: reclaiming Semaphore resource: Key: 0x68871d75 Sem Index: 1
Process 24436 not found: reclaiming Semaphore resource: Key: 0x68871d75 Sem Index: 2
```

6.6 Transport Broker

With the UMQ product, you use the '**broker**' transport to send messages from a source to a Queuing Broker, or from a Queuing Broker to a receiver.

The broker's Transport Pacing is receiver-paced, but only between the source and the broker. There is no end-to-end pacing with queuing.

When sources or receivers connect to a Queuing Broker, you must use the '**broker**' transport. You cannot use the '**broker**' transport with UMS or UMP products.

Chapter 7

Topic Resolution Description

Topic Resolution ("TR") is a set of protocols and algorithms used internally by Ultra Messaging to establish and maintain shared state. Here are the basic functions of TR:

- Receiver discovery of sources.
- DRO route maintenance and distribution.
- Persistent Store name resolution.
- Redundancy.

UM performs TR automatically; there are no API functions specific to normal TR operation. However, you can influence topic resolution by configuration. Moreover, you can set configuration options differently for individual topics, either by using **XML Configuration Files** (the <**topic**> element), or by using the API functions for setting configuration options programmatically (e.g. **lbm_rcv_topic_attr_setopt()** and **lbm_src_topic_attr_setopt()**). See UDP Topic Resolution Configuration Options for details.

An important design point of Topic Resolution is that information related to sources is distributed to all contexts in a UM network. This is done so that when a receiver object is created within a context, it can discover sources for the topic and join those sources. In support of this discovery process, each context maintains a memory-based *"resolver cache"*, which stores source information. The TR protocols and algorithms are largely in support of maintaining each context's resolver cache.

Topic Resolution also occurs across a DRO, which means between Topic Resolution Domains (TRDs). A receiver in one TRD will discover a source in a different TRD, potentially across many DRO hops. In this case, the DROs actively assist in TR. I.e. the sources and receivers in different TRDs do not exchange TR with each other directly, but rather with the assistance of the DRO.

Note

> With the UMQ product, Topic Resolution does not apply to brokered queuing sources, receivers, or the brokers themselves. However, ULB queuing does make use of topic resolution.

There are three different possible protocols used to provide Topic Resolution:

- Multicast UDP TR,
- Unicast UDP TR (with "lbmrd" service),
- TCP TR (with "SRS" service).

Of those three, Multicast UDP and Unicast UDP are mutually exclusive. It is not possible to configure UM to use both within a single TRD. Multicast is generally preferred over Unicast, with Unicast being selected when there are policy or environment reasons to avoid Multicast (e.g. cloud computing).

TCP-based TR (with "SRS" service) is a more recent addition to UM. It supports source discovery, and tracking of receivers.

TCP-based TR is often paired with one of the UDP-based TR protocols (multicast or unicast). This is done to support interoperability with pre-6.12 versions of UM, and supply TR functionality not yet available in TCP TR. The TCP-based and UDP-based TR protocols run in parallel, with the UDP-based TR protocol supporting interoperability with pre-6.12 components, and supplying the functionality missing from TCP TR.

When all of your UM components are upgraded to UM 6.14 and beyond, you can use the **resolver_disable_udp↵ _topic_resolution (context)** configuration option to turn off UDP-based TR.

The advantage of TCP-based TR is greater reliability and reduced network and CPU load. UDP-based TR is susceptible to deafness issues due to transient network failures. Avoiding those deafness issues requires configuring UDP-based TR to use significant network and CPU resources. In contrast, TCP-based TR is designed to be reliable with much less network and CPU load, even in the face of transient network failures.

7.1 Resolver Caches

Independent of the TR protocol used, a context maintains two resolver caches: a source-side cache and a receiver-side cache.

7.1.1 Source Resolver Cache

The source-side cache holds information about sources that the application created. It is used primarily by the context to respond to TR Queries.

7.1.2 Receiver Resolver Cache

The receiver-side cache holds information about *all* sources in the UM network:

- Sources in the current application. For example, if **Monitoring** is turned on and configured for the same TRD as the context being monitored, the source used to publish the monitoring data is included.
- Sources in local applications, in the same TRD.
- Sources in remote applications that are reachable by a DRO network.

Thus, the receiver-side cache can become large. In very large deployments, it may be necessary to increase the size of the receiver cache using **resolver_receiver_map_tablesz (context)**.

A context's receiver-side cache also holds information about receivers created by the current application. This is used by the context when TR Advertisements are received to assist in completing subscriptions.

Be aware that with UDP-based Topic Resolution, receiver-side cache entries are typically not removed when the corresponding sources are deleted, unless they are subscribed. I.e. if a source is created (but not subscribe) and then deleted and then re-created, there will be two entries in the cache: one for the old source and one for the new. For system designs that feature short-lived sources, the topic cache can grow over time without bound. In contrast, with TCP-based Topic Resolution, receiver-side cache entries typically are removed when the sources are deleted, even if not subscribed.

7.2 TR Protocol Comparison

7.2.1 Multicast UDP TR

Multicast UDP-based Topic Resolution is the default protocol.

Advantages:

- Very fast source discovery for small deployments.
- Simplicity – no independent service (daemon) required.
- Highly fault tolerant.

Disadvantages:

- As the number of topics grows, the speed of source discovery degrades and resource consumption increases (network bandwidth and CPU load). This resource consumption can introduce significant latency outliers (jitter).
- Since UDP is not a reliable protocol, Multicast UDP TR relies on repetition to ensure delivery of TR information.
- To effectively avoid deafness issues, resources must be consumed over the long term (TR must be configured to run "forever"). Jitter can be a long-term problem.
- As deployments change and grow, TR performance should be monitored and analyzed for possible reconfiguration to strike the right balance between speed of source discovery vs. resource consumption.
- By default, when sources are deleted, receivers are not informed unless *all* sources on a given transport session are deleted. Even if "final advertisements" are enabled, their delivery is best effort and not guaranteed.

See UDP-Based Topic Resolution Details for more information.

7.2.2 Unicast UDP TR

Unicast UDP-based Topic Resolution is functionally identical to Multicast UDP. It is used as a replacement for Multicast UDP in environments where the use of multicast is not possible (e.g. the cloud) or is against policy. The "lbmrd" service *simulates* multicast by simply forwarding all TR traffic to all contexts registered in a TRD. Note that the "lbmrd" service does not maintain state about the sources and receivers. It simply fans out Unicast TR.

Advantages:

- Does not use multicast (an advantage if multicast cannot be used).

Disadvantages:

- All the same disadvantages of Multicast UDP.
- Requires one or more independent "lbmrd" services, which should be monitored for failure and restarted.
- Due to fan-out, puts a greater load on network hardware.
- By default, when sources are deleted, receivers are not informed unless *all* sources on a given transport session are deleted. Even if "final advertisements" are enabled, their delivery is best effort and not guaranteed.

See UDP-Based Topic Resolution Details for more information.

7.2.3 TCP TR

TCP-based Topic Resolution is a newer implementation of a service-based distribution of source and receiver information. It is available as of UM version 6.12, in which it provides a subset of the total TR functionality. In a future UM version, TCP-based TR will provide *all* TR functionality, at which point it can be used to the exclusion of UDP-based TR. Until that time, TCP-based TR is typically paired with UDP-based TR (either Multicast or Unicast).

Advantages:

- Can allow UDP-based TR to be disabled or "dialed-back". I.e. its configuration can be adjusted to consume fewer CPU and network resources. See TCP-Based TR Version Interoperability.
- Since TCP is a reliable protocol, TCP-based TR does not need to repeatedly send the same information to ensure its reception.
- It is not necessary to consume resources over the long term to avoid deafness issues.
- If a source is deleted, that deletion is reliably communicated to all contexts in the TRD.

Disadvantages:

- For TRDs containing UM versions both before and after UM 6.12, TCP-based TR must be combined with UDP-based TR to support inter-version interoperability.
- For UM version 6.12 and 6.13, TCP-based TR does not fulfill the TR functions of DRO route maintenance or persistent Store name resolution. Users who require one or more of those functions should upgrade to UM version 6.14 or beyond.

Most users who combine UDP and TCP TR should be able to gradually reduce the CPU and Network load from UDP-based TR as the applications are upgraded to UM 6.12 and beyond.

When all of your UM components are upgraded to UM 6.14 and beyond, you can use the **resolver_disable_udp↩ _topic_resolution (context)** configuration option to turn off UDP-based TR.

See TCP-Based Topic Resolution Details for more information.

7.3 TCP-Based Topic Resolution Details

TCP-based TR was introduced in UM version 6.12 to address shortcomings in UDP-based TR:

- **Limit on scaling.** It is difficult to configure UDP-based TR to scale to many hundreds of thousands of topics. Too many topics typically results in unacceptable CPU and network load, and latency outliers (jitter). Intense TR bursts can cause packet loss, retransmissions, and deafness. As deployments grow in size and complexity, UDP-based TR typically requires greatly extended Sustaining Phases, often to infinity. This results in significant CPU and network resources over the long term, and introduces latency outliers (jitter).
- **High time to resolve.** To reduce the CPU and network load, and to avoid packet loss, UDP-based TR is usually strongly rate limited. This can greatly extend the time required to resolve topics, sometimes into the tens of minutes.

TCP-based TR differs from UDP-based TR in two important ways:

- With TCP-based TR, the TCP protocol ensures reliable transmission of information. TCP also makes use of congestion control algorithms to avoid packet loss.
- With TCP-based TR, topic information is maintained in the Stateful Resolution Service (SRS).

The basic approach used by TCP-based TR is as follows: Each context in a TRD is configured with the address of one or more SRS instances (up to 5). For fault-tolerance, two or three is typical. When the context is created, it connects to the configured SRSes. When the connection is successful, the context and SRSes exchange TR information. They normally do this without involving the other contexts in the TRD.

Then, as an application creates or deletes sources, its context informs the SRSes of the change, which in turn inform the other contexts in the TRD. In addition (as of UM 6.13), as an application creates or deletes receivers, the SRSes track that receiver interest. The SRSes do not distribute receiver interest to other applications, but rather use it to optimize the distribution of source information. An SRS only informs a context of sources that the context is interested in (has receiver for).

There are periodic handshakes between each context and the SRSes to ensure that connectivity is maintained and that state is valid. This removes the need to re-send TR information that has already been sent.

If an application loses connection with an SRS (perhaps due an extended network outage, or due to failure of the SRS), the context will repeatedly try to reconnect. Once successful, the process of exchanging TR information is repeated.

Note that much of the difficulty of configuring UDP-based TR is related to controlling the repeated transmission of the same TIRs and TQRs. With TCP-based TR, that repetition is eliminated, making both the configuration and the operation more straight-forward.

A note about the term "stateful" in relation to the SRS. Even though Unicast UDP TR uses a service called "lbmrd", that service does not maintain the topic information. The "lbmrd" is not "stateful". Instead, it merely forwards TR datagrams it receives, essentially simulating Multicast.

In contrast, the SRS maintains knowledge of all sources and receivers in the TRD (hence the "Stateful" in SRS). For a newly-started receiving application to discover an existing source, the SRS can send the information without the source getting involved.

With TCP-based TR, source advertisement messages are called "SIRs" (Source Information Records). This term is used elsewhere in the documentation.

For configuration information, see **TCP-Based Resolver Operation Options**.

Attention

> It is important to configure the SRS with SRS Element "<interface>", even if the SRS is running on a single-homed host (i.e. one network connection). Also, applications that connect to the SRS must configure their interfaces properly. See **default_interface (context)**.

Note

> The Network Address Translation (NAT) functionality provided by the lbmrd is not available with the SRS. If you want to mix SRS and lbmrd operation on the same network, `contact Informatica Support`.

7.3.1 TCP-Based TR and Fault Tolerance

As of UM version 6.13 and beyond, TCP-based TR supports redundancy. This is accomplished by starting two or more instances of the Stateful Resolver Service (SRS), typically on separate physical hosts, and configuring application and daemon contexts to connect to all of them. Although up to 5 SRSes can be configured, 2 or 3 are typical.

A context uses the **resolver_service (context)** option to configure the desired SRSes. Each context will establish TCP connections to all of the configured SRSes. The SRSes are used "hot/hot", with all information being sent to all configured SRSes, so there is no loss of Topic Resolution service if an SRS fails.

7.3.2 TCP-Based TR Interest

As of UM version 6.13, the SRS tracks topic interest of contexts. If an application creates a receiver for topic "XYZ", the context informs the SRS that it is interested in that topic. This allows the SRS to filter the TR traffic it sends to contexts, which greatly increases the scalability of TR.

The SRS only sends source advertisements to contexts that are interested in that source's topic. Contexts also inform the SRS of wildcard receivers, in which case the SRS will send source advertisements for all sources that match the topic pattern.

Warning

> While SRS filtering has the benefit of TR reducing traffic to a context, it can interfere with the **resolver_source_notification_function (context)** and resolver_event_function features. Some applications use those features to inform the application of the availability of sources *before* receivers are created. But the SRS will normally not inform the context of sources for which the context has no receiver. For applications that require these features, filtering must be turned off by setting **resolver_service_interest_mode (context)** to "flood".

7.3.3 TCP-Based TR Version Interoperability

TCP-based TR was first introduced in UM version 6.12. To maintain interoperability with pre-6.12, applications can be configured to use both TCP and UDP-based TR, in parallel.

This can make it difficult to gain all the benefits of TCP-based TR. Since pre-6.12 applications still need to avoid the problems of deafness, even applications that have upgraded to 6.12 and beyond need to enable UDP-based TR, usually with extended sustaining phases, often to infinity.

Ideally, all applications within a TRD can be upgraded to 6.14 and beyond, eliminating the need for UDP-based TR, but this is often not practical. How can the TR load be reduced in a step-wise fashion while an organization is upgrading applications gradually, over a long period of time?

Fortunately, You can set configuration options differently for individual topics, either by using **XML Configuration Files** (the <**topic**> element), or by using the API functions for setting configuration options programmatically (e.g. **lbm_rcv_topic_attr_setopt()** and **lbm_src_topic_attr_setopt()**).

Some helpful strategies might be:

- Identify those topics or classes of topics that have limited application interest. If topic X has sources and receivers in upgraded applications, the UDP-based TR for that topic can be reduced (e.g. sustaining phase greatly reduced).

- Identify those TRDs that have small numbers of applications. When a given TRD's applications have all been upgraded, the UDP-based TR for *all* topics in that TRD can be reduced. If practical, applications can be moved between TRDs to enable some TRDs to be populated by UM version 6.12 and beyond. Also, a TRD can be sub-divided, separating pre-upgraded from post-upgraded.

When all of your UM components are upgraded to UM 6.14 and beyond, you can use the **resolver_disable_udp_topic_resolution (context)** configuration option to turn off UDP-based TR.

7.3.4 TCP-Based TR Configuration

A UM context is configured to use TCP-based TR with the option **resolver_service (context)**, which tells how to connect to the SRS. For example:

```
context resolver_service 10.29.3.41:12000
```

A DNS host name can be used instead of an IP address:

```
context resolver_service test1.informatica.com:12000
```

For fault tolerance, more than one running SRS instance can be configured:

```
context resolver_service test1.informatica.com:12000,test2.informatica.com:12000
```

This assumes that an SRS service is running at that address:port.

If interoperability with UDP-based TR is not needed, UM should be configured to use ONLY the SRS for topic resolution using the option **resolver_disable_udp_topic_resolution (context)**.

7.3.5 SRS Service

The SRS service is a daemon process which must be run to provide TCP-based TR for a TRD.

See Man Pages for SRS for details on running the SRS service.

All the contexts in the TRD must be configured to connect to the SRS with the option **resolver_service (context)**. After connecting, each context exchanges TR information with the SRS.

Attention

> Applications that connect to the SRS must configure their interfaces properly. See **default_interface (context)**.

As applications create and delete sources, the SRS is informed, and the SRS informs all connected contexts. This includes proxy sources from a DRO. In addition, a periodic "keepalive" handshake is performed between the SRS and all connected contexts.

If a network failure causes the context's connection to the SRS to be broken, the context will periodically retry the connection. Since most network failures are brief, the context will soon successfully re-establish a connection to the SRS. Even though this is a resumption of the same context's earlier connection, the context and SRS still exchange full TR information to make sure that any changes during the disconnected period are reflected.

The SRS also supports the publishing of operational and status information via the daemonstatistics feature. For full details on the SRS Daemon Statistics details, see SRS Daemon Statistics.

SRS State Lifetime

If an application exits abnormally, the SRS will detect that the TCP connection is broken. However, the SRS must not *assume* that the application has failed; it might be a network problem that forced the disconnection.

So the SRS flags all sources owned by that context as "potentially down", and starts a "source state lifetime" timer (see <source-state-lifetime>). If the context has *not* failed, and reconnects within that period, during the initial exchange of TR information, the SRS will unflag any "potentially down" sources. However, in the case of application failure, when the state lifetime expires, all "potentially down" sources are deleted. All connected contexts are informed of those deletions.

Note that as of UM version 6.13, the SRS also tracks application interest (topics for which the context has receivers). This interest is also remembered by the SRS if the connection is broken, and also has an "interest state lifetime" timer (see <interest-state-lifetime>). If the context has *not* failed, and reconnects within that period, during the initial exchange of TR information, the SRS will unflag any "potentially down" receiver interest. However, in the case of application failure, when the state lifetime expires, all "potentially down" receiver interest is deleted.

To maintain compatibility with 6.12 configurations, the SRS Element "<state-lifetime>" is maintained, and is used as the default value for both source and interest state lifetimes.

Note that if an application fails and then restarts, its connection to the SRS is *not* considered to be a resumption of the previous connection. It is considered to be a new context, and any sources created are new sources. The previous application instance's sources will remain in the "potentially down" state, and will time out with the state lifetime.

If a network outage lasts longer than the configured state lifetime, the SRS gives up on the context and deletes sources and interest. These deletions are communicated to all connected contexts. When the network outage is repaired and the context reconnects, the exchange of TR information with the SRS will re-create the context's sources and interest in the SRS, and communicate them to other contexts. This restores normal operation.

As of UM version 6.14, the SRS also tracks routing information sent by the DRO. That information is handled in much the same way as client information.

SRS Log File

The SRS generates log messages that are used to monitor its health and operation. You can configure these to be directed to "console" (standard output) or a specified log "file", via the <log> configuration element. Normally "console" is only used during testing; a persistent log file should be used for production. The SRS does not overwrite its log files on startup, but instead appends to it.

SRS Rolling Logs

To prevent unbounded disk file growth, the SRS 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 SRS daemon process.
- *DATE* - Date that the log file was rolled, in YYYY-MM-DD format.
- *SEQNUM* - Sequence number, starting at 1 when the process starts, and incrementing each time the log file rolls.

For example: `srs.log_9867.2017-08-20.2`

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

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

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

Let's say that the log file grows at 1 million bytes per hour (VERY unlikely for an SRS, but let's assume for illustration purposes). 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.

In addition, the SRS supports automatic deletion of log files based on either or both of two criteria: max history, and total size cap. The max history refers to the number of archived log files, and the total size cap refers to the sum of the sizes of the archived files in millions of bytes. When either or both criteria are met, one or more of the oldest log files are removed until the criteria no longer apply.

For more information, see the <log> configuration element.

7.4 SRS 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 SRSes. This monitoring should include real-time automated detection of problems that will produce a timely alert to operations staff.

Two types of data should be monitored:

- Log file.
- Daemon statistics.

7.4.1 SRS Monitoring: Logs

Ideally, log file monitoring would support the following:

- Archive all log messages for all SRSes for at least a week, preferably a month.
- Provide rapid access to operations staff to view the latest log messages from a SRS.
- Periodic scans of the log file to detect errors and raise alerts to operations staff.

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

```
Tue Nov 1 13:29:37 CDT 2022 [INFO]: SRS-10385-99: TCP Disconnect received for
    client[10.29.3.101:47205] sessionID[0x8295ab9a]
```

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.

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

7.4.2 SRS Monitoring: Daemon Stats

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

- Protobufs - recommended.
- JSON - deprecated. Informatica recommends migrating to protobufs.

For information on the deprecated JSON formatted daemon stats, see SRS Daemon Statistics.

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

For example, here is an excerpt from a sample SRS configuration file that shows how its daemon stats can be published:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  ...
```

```
<daemon-monitor topic="/29west/statistics">
  <ping-interval>600000</ping-interval>
  <publishing-interval>
    <default>600000</default>
  </publishing-interval>
  <lbm-attributes>
    <option scope="context" name="resolver_unicast_daemon" value="10.29.3.101:12801"/>
    <option scope="context" name="default_interface" value="10.29.3.0/24"/>
    <option scope="context" name="mim_incoming_address" value="0.0.0.0"/>
    <option scope="source" name="transport" value="tcp"/>
    ...
  </lbm-attributes>
  <monitor-format>pb</monitor-format>
</daemon-monitor>
```

Notes:

1. The SRS Element "<monitor-format>" value "pb" selects the protobuf format and is available for the SRS in UM version 6.15 and beyond.
2. For a list of possible protobuf messages for the SRS, see the "srs_mon.proto" file at **Example srs_mon.proto**.
3. The SRS Element "<ping-interval>" and SRS Element "<default>" define 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. The LBM attributes included in the above example within the SRS Element "<daemon-monitor>" element 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`. There are other ways to specify the interface; see **Specifying Interfaces**.
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`

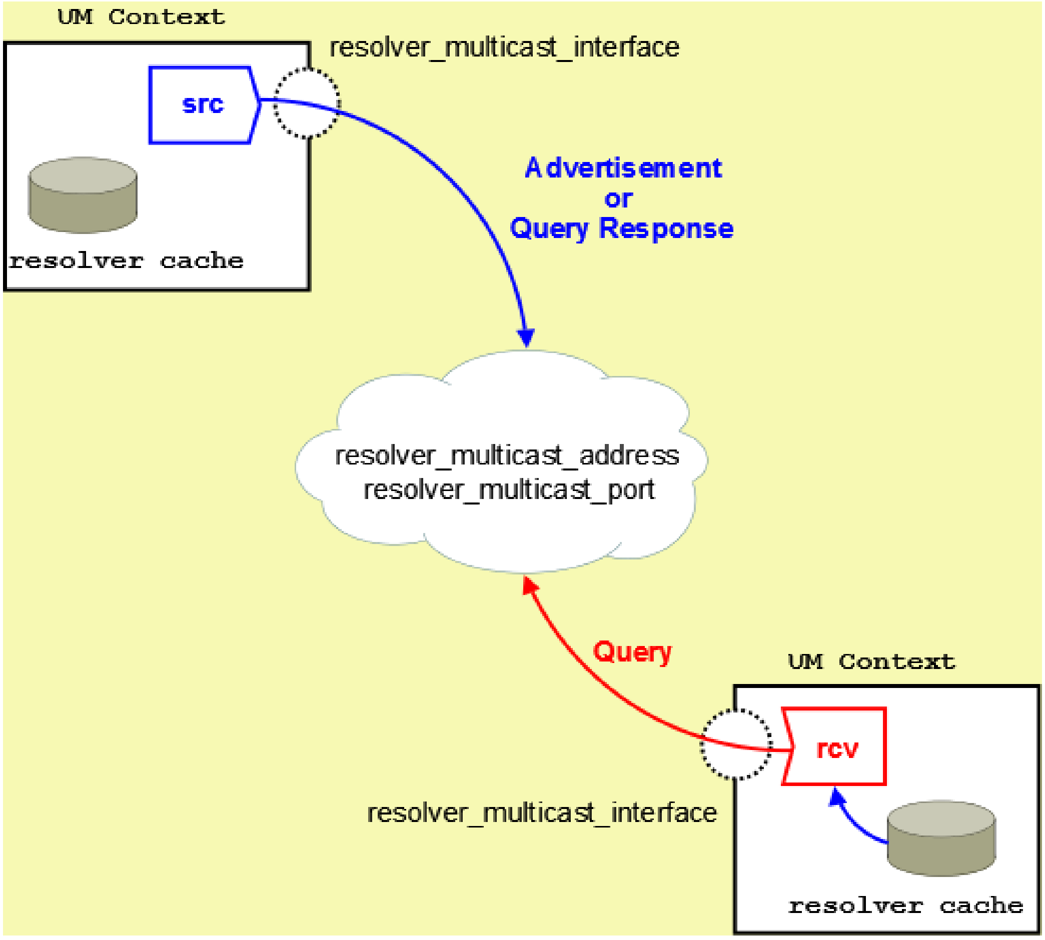
Chapter 8

Architecture

UM is designed to be a flexible architecture. Unlike many messaging systems, UM does not require an intermediate daemon to handle routing issues or protocol processing. This increases the performance of UM and returns valuable computation time and memory back to applications that would normally be consumed by messaging daemons.

8.1 UDP-Based Topic Resolution Details

The following diagram illustrates UDP-based Topic Resolution. The diagram references multicast configuration options, but the concepts apply equally to unicast.



By default, Ultra Messaging relies on UDP-based Topic Resolution. UDP-based TR uses *queries* (TQRs) and *advertisements* (TIRs) to resolve topics. These TQRs and TIRs are sent in UDP datagrams, typically with more than one TIR or TQR in a given datagram.

UDP-based topic resolution traffic can benefit from hardware acceleration. See **Transport Acceleration Options** for more information.

For Multicast UDP, TR datagrams are sent to an IP multicast group and UDP port configured with the Ultra Messaging configuration options **resolver_multicast_address (context)** and **resolver_multicast_port (context)**).

For Unicast UDP, TR datagrams are sent to the IP address and port of the "lbmrd" daemon. See the UM configuration option **resolver_unicast_daemon (context)**.

Note that if both Multicast and Unicast are configured, the Unicast has higher precedence, and Multicast will not be used.

UDP-based Topic Resolution occurs in the following phases:

- Initial Phase - Period that allows you to resolve a topic aggressively. This phase can be configured to run differently from the defaults or completely disabled.
- Sustaining Phase - Period that allows new receivers to resolve a topic after the Initial Phase. Can also be the primary period of topic resolution if you disable the Initial Phase. This phase can also be configured to run differently from the defaults or completely disabled.
- Quiescent Phase - The quiet phase where Topic Resolution datagrams are no longer sent in an unsolicited way. This reduces the CPU and network resources consumed by TR, and also reduces latency outliers (jitter). However, in large deployments, especially those that include wide-area networks, the Quiescent Phase is sometimes disabled, by configuring the Sustaining Phase to continue forever. This is done to avoid deafness issues.

The phases of topic resolution are specific to individual topics. A single context can have some topics in each of the three phases running concurrently.

8.1.1 Sources Advertise

For UDP-based TR, Sources use Topic Resolution in the following ways:

- Unsolicited advertisement of active sources. When a source is first created, it enters the Initial Phase of TR. During the Initial, and subsequent Sustaining phases, the source sends Topic Information Record datagrams (TIRs) to all the other contexts in the TRD. The source does this in an unsolicited manner; it advertises even if there are no receivers for its topic.
- Respond to Topic Queries. When a receiver is first created, it enters the Initial phase of TR. During the Initial, and subsequent Sustaining phases, the receiver sends Topic Query Records (TQRs) to all other contexts in the TRD. When a source receives a TQR for its topic, it will restart its Sustaining Phase of advertising to ensure that the receiver discovers the source.

A TIR contains all the information that the receiver needs to join the topic's Transport Session. The TIR datagram sent unsolicited is identical to the TIR sent in response to a TQR. Depending on the transport type, a TIR will contain one of the following groups of information:

- For transporttcp, the source address, TCP port and Session ID.
- For Transport LBT-RM, the source address, the multicast group address, the UDP destination port, LBT-RM Session ID, and the unicast UDP port to which NAKs are sent.
- For Transport LBT-RU, the source address, UDP port and Session ID.

- For Transport LBT-IPC, the Host ID, LBT-IPC Session ID and Transport ID.
- For transportlbtsmx, the Host ID, LBT-SMX Session ID and Transport ID.

See **UDP-Based Resolver Operation Options** for more information.

8.1.2 Receivers Query

For UDP-based TR, when an application creates a receiver within a context, the new receiver first checks the context's resolver cache for any matching sources that the context has already discovered. Those will be joined immediately.

In addition, the receiver normally initiates a process of sending Topic Query Records (TQRs). This triggers sources for the receiver's topic to advertise, if they are not already. This allows sources which are in their Quiescent Phase to be discovered by new receivers.

A TQR consists primarily of the topic string.

8.1.3 Wildcard Receiver Topic Resolution

For UDP-based TR, UM Wildcard Receivers use Topic Resolution in conceptually the same ways as a single-topic receiver, although some of the details are different. Instead of searching the resolver cache for a specific topic, a new wildcard receiver object searches for all sources that match the wildcard pattern.

Also, the TQRs contain the wildcard pattern, and all sources matching the pattern will advertise.

Finally, wildcard receivers omit the Sustaining Phase for sending Queries. They only support Initial and Quiescent Phases.

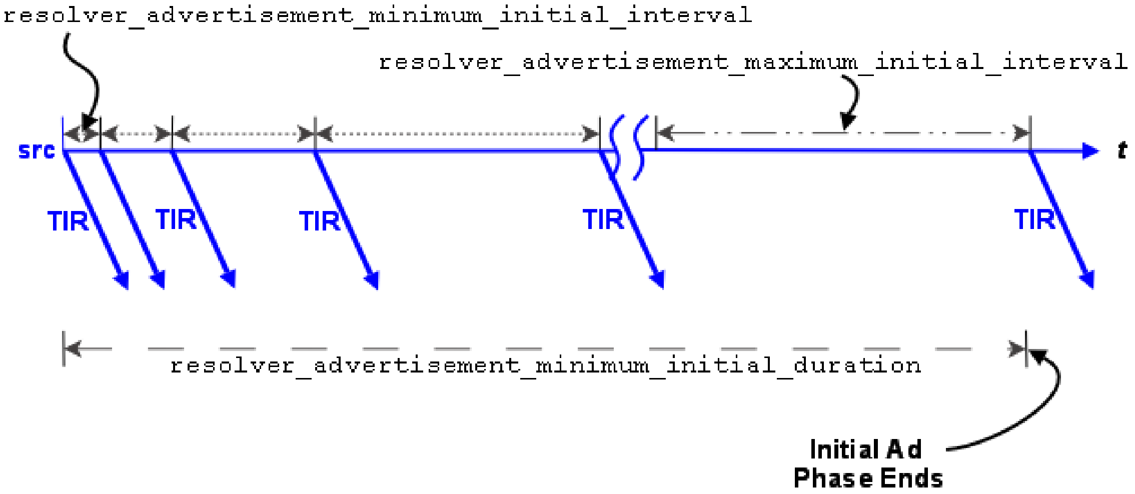
See **Wildcard Receiver Options** for more information.

8.1.4 Initial Phase

For UDP-based TR, the initial topic resolution phase for a topic is an aggressive phase that can be used to resolve all topics before sending any messages. During the initial phase, network traffic and CPU utilization might actually be higher. You can completely disable this phase, if desired. See **Disabling Aspects of Topic Resolution** for more information.

Advertising in the Initial Phase

For the initial phase default settings, the resolver issues the first advertisement as soon as the scheduler can process it. The resolver issues the second advertisement 10 ms later, or at the **resolver_advertisement_minimum_initial_interval (source)**. For each subsequent advertisement, UM doubles the interval between advertisements. The source sends an advertisement at 20 ms, 40 ms, 80 ms, 160 ms, 320 ms and finally at 500 ms, or the **resolver_advertisement_maximum_initial_interval (source)**. These 8 advertisements require a total of 1130 ms. The interval between advertisements remains at the maximum 500 ms, resulting in 7 more advertisements before the total duration of the initial phase reaches 5000 ms, or the **resolver_advertisement_minimum_initial_duration (source)**. This concludes the initial advertisement phase for the topic.

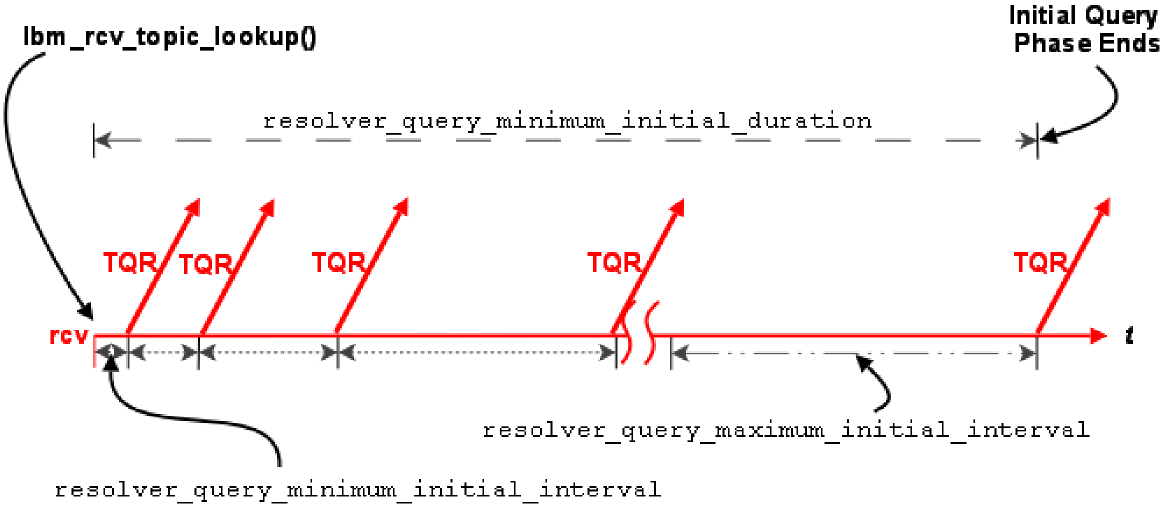

resolver_advertisement_minimum_initial_interval
resolver_advertisement_maximum_initial_interval
src
t
TIR
TIR
TIR
TIR
TIR
resolver_advertisement_minimum_initial_duration
Initial Ad Phase Ends

The initial phase for a topic can take longer than the **resolver_advertisement_minimum_initial_duration (source)** if many topics are in resolution at the same time. The configuration options, **resolver_initial_advertisements_per_second (context)** and **resolver_initial_advertisement_bps (context)** enforce a rate limit on topic advertisements for the entire UM context. A large number of topics in resolution - in any phase - or long topic names may exceed these limits.

If a source advertising in the initial phase receives a topic query, it responds with a topic advertisement. UM recalculates the next advertisement interval from that point forward as if the advertisement was sent at the nearest interval.

Querying in the Initial Phase

Querying activity by receivers in the initial phase operates in similar fashion to advertising activity, although with different interval defaults. The **resolver_query_minimum_initial_interval (receiver)** default is 20 ms. Subsequent intervals double in length until the interval reaches 200 ms, or the **resolver_query_maximum_initial_interval (receiver)**. The query interval remains at 200 ms until the initial querying phase reaches 5000 ms, or the **resolver_query_minimum_initial_duration (receiver)**.


lbm_rcv_topic_lookup()
Initial Query Phase Ends
resolver_query_minimum_initial_duration
TQR
TQR
TQR
TQR
TQR
rcv
t
resolver_query_maximum_initial_interval
resolver_query_minimum_initial_interval

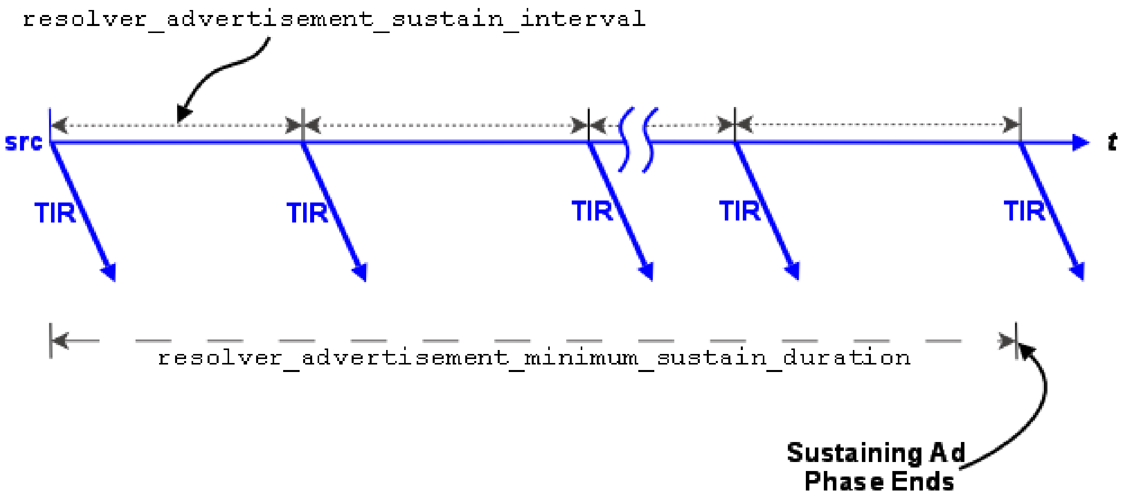
The initial query phase completes when it reaches the **resolver_query_minimum_initial_duration (receiver)**. The initial query phase also has UM context-wide rate limit controls (**resolver_initial_queries_per_second (context)** and **resolver_initial_query_bps (context)**) that can result in the extension of a phase's duration in the case of a large number of topics or long topic names.

8.1.5 Sustaining Phase

For UDP-based TR, the sustaining topic resolution phase follows the initial phase and can be a less active phase in which a new receiver resolves its topic. It can also act as the sole topic resolution phase if you disable the initial phase. The sustaining phase defaults use less network resources than the initial phase and can also be modified or disabled completely. See **Disabling Aspects of Topic Resolution** for details.

Advertising in the Sustaining Phase

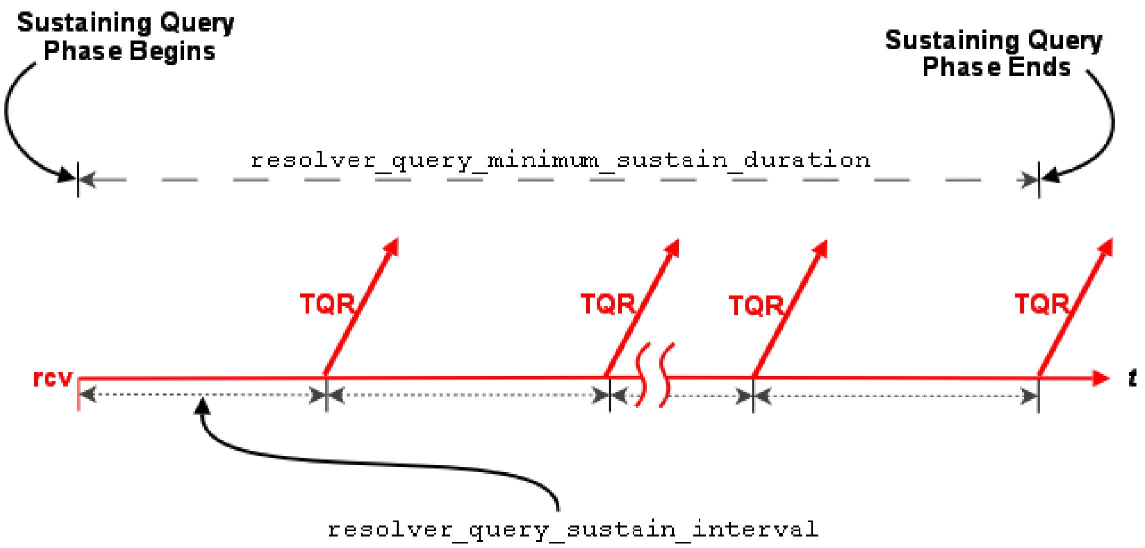
For the sustaining phase defaults, a source sends an advertisement every second (**resolver_advertisement_↩ sustain_interval (source)**) for 1 minute (**resolver_advertisement_minimum_sustain_duration (source)**). When this duration expires, the sustaining phase of advertisement for a topic ends. If a source receives a topic query, the sustaining phase resumes for the topic and the source completes another duration of advertisements.


resolver_advertisement_sustain_interval
src
t
TIR
TIR
TIR
TIR
TIR
resolver_advertisement_minimum_sustain_duration
Sustaining Ad Phase Ends

The sustaining advertisement phase has UM context-wide rate limit controls (**resolver_sustain_advertisements↩ _per_second (context)** and **resolver_sustain_advertisement_bps (context)**) that can result in the extension of a phase's duration in the case of a large number of topics or long topic names.

Querying in the Sustaining Phase

Default sustaining phase querying operates the same as advertising. Unresolved receivers query every second (**resolver_query_sustain_interval (receiver)**) for 1 minute (**resolver_query_minimum_sustain_duration (receiver)**). When this duration expires, the sustaining phase of querying for a topic ends.


Sustaining Query Phase Begins
Sustaining Query Phase Ends
resolver_query_minimum_sustain_duration
TQR
TQR
TQR
TQR
rcv
t
resolver_query_sustain_interval

Sustaining phase queries stop when one of the following events occurs:

- The receiver discovers multiple sources that equal **resolution_number_of_sources_query_threshold (receiver)**.
- The sustaining query phase reaches the **resolver_query_minimum_sustain_duration (receiver)**.

The sustaining query phase also has UM context-wide rate limit controls (**resolver_sustain_queries_per_second (context)** and **resolver_sustain_query_bps (context)**) that can result in the extension of a phase's duration in the case of a large number of topics or long topic names.

8.1.6 Quiescent Phase

For UDP-based TR, this phase is the absence of topic resolution activity for a given topic. It is possible that some topics may be in the quiescent phase at the same time other topics are in initial or sustaining phases of topic resolution.

This phase ends if either of the following occurs.

- A new receiver sends a query.
- Your application calls **lbm_context_topic_resolution_request()** that provokes the sending of topic queries for any receiver or wildcard receiver in this state.

8.1.7 Store (context) Name Resolution

For UDP-based TR, with the UMP/UMQ products, topic resolution facilitates the resolution of persistent Store names to a DomainID:IPAddress:Port.

Topic Resolution resolves Store (or context) names by sending context name queries and context name advertisements over the topic resolution channel. A Store name resolves to the Store's DomainID:IPAddress:Port. You configure the Store's name and IPAddress:Port in the Store's XML configuration file. See **Identifying Persistent Stores** for more information.

If you do not use the DRO, the DomainID is zero. Otherwise, the DomainID represents the Topic Resolution Domain where the Store resides. Stores learn their DomainID by listening to Topic Resolution traffic.

Via the Topic Resolution channel, sources query for Store names and Stores respond with an advertisement when they see a query for their own Store name. The advertisement contains the Store's DomainID:IPAddress:Port.

For a new source configured to use Store names (**ume_store_name (source)**), the resolver issues the first context name query as soon as the scheduler can process it. The resolver issues the second advertisement 100 ms later, or at the **resolver_context_name_query_minimum_interval (context)**. For each subsequent query, UM doubles the interval between queries. The source sends a query at 200 ms, 400 ms, 800 ms and finally at 1000 ms, or the **resolver_context_name_query_maximum_interval (context)**. The interval between queries remains at the maximum 1000 ms until the total time querying for a Store (context) name equals **resolver_context_name_query_duration (context)**. The default for this duration is 0 (zero) which means the resolver continues to send queries until the name resolves. After a Store name resolves, the resolver stops sending queries.

If a source sees advertisements from multiple Stores with the same name, or a Store sees an advertisement that matches its own Store name, the source issues a warning log message. The source also issues an informational log message whenever it detects that a resolved Store (context) name changes to a different DomainID:IPAddress:Port.

8.1.8 UDP Topic Resolution Configuration Options

See the following sections in `UM Configuration Guide` for more information:

- **UDP-Based Resolver Operation Options**
- **Multicast Resolver Network Options**
- **Unicast Resolver Network Options**
- **Wildcard Receiver Options**

Assigning Different Configuration Options to Individual Topics

You can set configuration options differently for individual topics, either by using **XML Configuration Files** (the <**topic**> element), or by using the API functions for setting configuration options programmatically (e.g. **lbm_rcv_topic_attr_setopt()** and **lbm_src_topic_attr_setopt()**).

8.1.9 Unicast UDP Topic Resolution

By default UM expects multicast connectivity between all sources and receivers. When only unicast connectivity is available, you may configure all sources and receivers to use unicast topic resolution. This requires that you run one or more instances of the UM unicast topic resolution daemon (lbmrd), which perform the same topic resolution activities as multicast topic resolution. You configure your applications to use the lbmrd daemons with **resolver_unicast_daemon (context)**.

See Lbmrd Man Page for details on running the lbmrd daemon.

The lbmrd can run on any machine, including the source or receiver. Of course, sources will also have to select a transport protocol that uses unicast addressing (e.g. TCP, TCP-LB, or LBT-RU). The lbmrd maintains a table of clients (address and port pairs) from which it has received a topic resolution message, which can be any of the following:

- Topic Information Records (TIR) - also known as topic advertisements

- Topic Query Records (TQR)
- keepalive messages, which are only used in unicast topic resolution

After lbmrd receives a TQR or TIR, it forwards it to all known clients. If a client (i.e. source or receiver) is not sending either TIRs or TQRs, it sends a keepalive message to lbmrd according to the **resolver_unicast_keepalive_interval (context)**. This registration with the lbmrd allows the client to receive advertisements or queries from lbmrd. lbmrd maintains no state about topics, only about clients.

LBMRD with the DRO Best Practice

If you're using the lbmrd for topic resolution across a DRO, you may want all of your domains discovered and all routes to be known before creating any topics. If so, change the UM configuration option, **resolver_unicast_↩ force_alive (context)**, from the default setting to 1 so your contexts start sending keepalives to lbmrd immediately. This makes your startup process cleaner by allowing your contexts to discover the other Topic Resolution Domains and establish the best routes. The trade-off is a little more network traffic every 5 seconds.

Unicast Topic Resolution Resilience

Running multiple instances of lbmrd allows your applications to continue operation in the face of a lbmrd failure. Your applications' sources and receivers send topic resolution messages as usual, however, rather than sending every message to each lbmrd instance, UM directs messages to lbmrd instances in a round-robin fashion. Since the lbmrd does not maintain any resolver state, as long as one lbmrd instance is running, UM continues to forward LBMR packets to all connected clients. UM switches to the next active lbmrd instance every 250-750 ms.

8.1.10 Network Address Translation (NAT)

For UDP-based TR, if your network architecture includes LANs that are bridged with Network Address Translation (NAT), UM receivers will not be able to connect directly to UM sources across the NAT. Sources send Topic Resolution advertisements containing their local IP addresses and ports, but receivers on the other side of the NAT cannot access those sources using those local addresses/ports. They must use alternate addresses/ports, which the NAT forwards according to the NAT's configuration.

The recommended method of establishing UM connectivity across a NAT is to run a pair of DROs connected with a single TCP peer link. In this usage, the LANs on each side of the NAT are distinct Topic Resolution Domains.

Alternatively, if the NAT can be configured to allow two-way UDP traffic between the networks, the lbmrd can be configured to modify Topic Resolution advertisements according to a set of rules defined in an XML configuration file. Those rules allow a source's advertisements forwarded to local receivers to be sent as-is, while advertisements forwarded to remote receivers are modified with the IP addresses and ports that the NAT expects. In this usage, the LANs on each side of the NAT are combined into a single Topic Resolution domain.

Note

> TCP-based Topic Resolution using the SRS does not support this form of NAT operation. If you want to mix SRS and lbmrd operation on the same network, `contact Informatica Support`.

Warning

> Using an lbmrd NAT configuration severely limits the UM features that can be used across the NAT. Normal source-to-receiver traffic is supported, but the following more-advanced UM features are not compatible with lbmrd NAT:
>
> - Persistence
> - Queuing
> - Request/Response
> - Late Join
> - Off-Transport Recovery (OTR)

- Sending to Sources
- Unicast Immediate Messaging (UIM)

Late Join, sending to sources, and OTR can be made to work if applications are configured to use the default value (0.0.0.0) for **request_tcp_interface (context)**. This means that you cannot use **default_interface (context)**. Be aware that the DRO requires a valid interface be specified for **request_tcp_interface (context)**. Thus, **lbmrd NAT support for Late Join, Request/Response, and OTR is not compatible with UM topologies that contain the DRO**.

8.1.11 Example NAT Configuration

In this example, there are two networks, A and B, that are interconnected via a NAT firewall. Network A has IP addresses in the 10.1.0.0/16 range, and B has IP addresses in the 192.168.1/24 range. The NAT is configured such that hosts in network B have no visibility into network A, and can send TCP and UDP packets to only a single host in A (10.1.1.50) via the NAT's external IP address 192.168.1.1, ports 12000 and 12001. I.e. packets sent from B to 192.168.1.1:12000 are forwarded to 10.1.1.50:12000, and packets from B to 192.168.1.1:12001 are forwarded to 10.1.1.50:12001. Hosts in network A have full visibility of network B and can send TCP and UDP packets to hosts in B by their local 192 addresses and ports. Those packets have their source addresses changed to 192.168.1.1.

Since hosts in network A have full visibility into network B, receivers in network A should be able to use source advertisements from network B without any changes. However, receivers in network B will not be able to use source advertisements from network A unless those advertisements' IP addresses are transformed.

The lbmrd is configured for NAT using its XML configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <interface>10.1.1.50</interface>
    <port>12000</port>
  </daemon>
  <domains>
    <domain name="Net-NYC">
      <network>10.1.0.0/16</network>
    </domain>
    <domain name="Net-NJC">
      <network>192.168.1/24</network>
    </domain>
  </domains>
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        <match address="10.1.1.50" port="*"/>
        <replace address="192.168.1.1" port="*"/>
      </rule>
    </transform>
  </transformations>
</lbmrd>
```

The lbmrd must be run on 10.1.1.50.

The application on 10.1.1.50 should be configured with:

```
context resolver_unicast_daemon 10.1.1.50:12000
source transport_tcp_port 12001
```

The applications in the 192 network should be configured with:

```
context resolver_unicast_daemon 192.168.1.1:12000
source transport_tcp_port 12100
```

With this, the application on 10.1.1.50 is able to create sources and receivers that communicate with applications in the 192 network.

See lbmrd Configuration File for full details of the XML configuration file.

8.2 UDP-Based Topic Resolution Strategies

Configuring UDP-based TR frequently involves a process of weighing the costs and benefits of different goals. The most common goals involved are:

- **Avoid "deafness".** deafness is when there is a source and a receiver for a topic, but the receiver does not discover the source. This is usually a very high priority goal.
- **Minimize the delay before a transport session is joined.** This is especially important when a new source is created and the application wants to wait until all existing receivers have fully joined the transport session before sending messages.
- **Minimizing impact on the system.** Sending and receiving TR datagrams consumes CPU, network bandwidth, and can introduce latency outliers (jitter) on active data transports.
- **Maximizing scalability and flexibility.** Some deployments are tightly-coupled, carefully controlled, and well-defined. In those cases, scalability and flexibility might not be high-priority goals. Other deployments are loosely-coupled, and consist of many different application groups that do not necessarily coordinate their use of UM with each other. In those cases, scalability and flexibility can be important.
- **Fault tolerance.** Some environments, especially those that include Wide Area Networks, can have periodic degradation or loss of network connectivity. It is desired that after a given network problem is resolved, UM will quickly and automatically reestablish normal operation without deafness.

The right TR strategy for a given deployment can depend heavily on the relative importance of these and other goals. It is impossible to give a "one size fits all" solution. Most users work with Informatica engineers to design a custom configuration.

Most users employ a variation on a few basic strategies. Note for the most part, these strategies do not depend on the specific UDP protocol (Multicast vs. Unicast). Normally Multicast is chosen, except where network or policy restrictions forbid it.

8.2.1 Default TR

The main characteristics of UM's default TR settings are:

- Multicast UDP.
- Three phases enabled (Initial, Sustaining, Quiescent). Unsolicited TIRs and TQRs nominally last for 65 seconds, although that number can grow as the number of sources or receivers in a context increases.

The default settings can be fine for reasonably small, static deployments, typically not including Wide Area Networks. (A "static" deployment is one where sources, and receivers are, for the most part, created during system startup, and deleted during system shutdown. Contrast with a "dynamic" system where applications come and go during normal operation, with sources and receivers being created and deleted at unpredictable times.)

Advantages:

- Simplicity.
- In a network where sources and receivers are relatively static, the consumption of resources by TR stops reasonably quickly.

Disadvantages:

- As the numbers of contexts, sources, and receivers grow, the traffic load during the initial phase can be very intense, leading to packet loss and potential deafness issues. In these cases, the initial phase can be configured to be less aggressive, or disabled altogether.

- If a network outage lasts longer than 65 seconds, it is possible for new sources and receivers to be deaf to each other, due to entering their quiescent phases. In these cases, the sustaining phase can be configured for longer durations.

8.2.2 Query-Centric TR

The main characteristics of Query-centric TR are:

- Unsolicited TIRs are severely limited or disabled. See **Disabling Aspects of Topic Resolution**.
- TQRs are extended, often to infinity.

Query-centric TR can be useful for large-scale, dynamic systems, especially those that may have many sources for which there are no receivers during normal operation. For example, in some market data distribution architectures, many tens of thousands of sources are created, but a fairly small percentage of them have receivers at any given time. In that case, it is unnecessary to advertise sources on topics that have no receivers.

Note that this strategy does not prevent advertisements. Each TQR will trigger one or more sources to send a TIR in response.

Advantages:

- For some deployments, can result in significantly reduced TR loading due to removal of TIRs for topics with no receivers.

Disadvantages:

- To avoid deafness issues, the Query sustaining phase is usually extended, often to infinity. This consumes CPU and Network bandwidth, and can introduce latency outliers (jitter).
- For topics that have receivers, both TQR and TIR traffic are present. (In contrast, a Advertise-Centric TR strategy removes the TQRs, but at the expense of advertising *all* sources, even those that have no receivers.)

8.2.3 Known Query Threshold TR

In a special case of Query-centric TR, certain classes of topics have a specific number of sources. For example, in point-to-point use cases, a particular topic has exactly one source. As another example, some market data distribution architectures have two sources for each topic, a primary and a warm standby.

For those topics where it is known how many sources there should be, the configuration option **resolution_↩ number_of_sources_query_threshold (receiver)** can be combined with Query-centric TR to great benefit. Configure the receiver to query forever, and configure the query threshold to the number of expected sources. When the receiver discovers at least that many sources for the topic, it suppresses queries. Note that if a source exits and the receiver detects EOS, the number of known sources decreases. If the number drops below the threshold, queries resume.

For example, consider a market data system with a primary and warm standby source for each topic. Unsolicited advertisements are disabled (see **Disabling Aspects of Topic Resolution**), and **resolution_number_of_sources↩ _query_threshold (receiver)** is set to 2. The receiver will query until it has discovered two sources, at which point it will stop sending queries. If a source fails, the receiver resumes sending queries until it again has two sources.

The advantage here is that it is no longer necessary to extend the Sustaining phase forever to avoid deafness.

NOTE: wildcard receivers do not fit well with this model of TR. Wildcard receivers have their own query mechanism; see Wildcard Receiver Topic Resolution. In particular, there is no wildcard equivalent to the number of sources query threshold. In a query-centric model, wildcard queries must be extended to avoid potential deafness issues. However, in most deployments, the number of wildcard receiver objects is small compared to the number of regular single-topic receivers, so using the Known Query Threshold TR model can still be beneficial.

8.2.4 Advertise-Centric TR

The main characteristics of Advertise-centric TR are:

- Unsolicited TQRs are severely limited or disabled. See **Disabling Aspects of Topic Resolution**.
- TIRs are extended, often to infinity.

Advertise-centric TR can be useful for large-scale, dynamic systems, especially those that may have very few sources for which there are no receivers. For example, most order management and routing systems use messaging in a point-to-point fashion, and every source should have a receiver. In that case, it is unnecessary to extend queries.

Advantages:

- For some deployments, can result in moderate reduced TR loading due to reduction of TQRs.

Disadvantages:

- To avoid deafness issues, the Advertising sustaining phase is usually extended, often to infinity. This consumes CPU and Network bandwidth, and can introduce latency outliers (jitter).
- For topics that have no receivers, TIR traffic is present. (In contrast, a Query-Centric TR strategy removes the TIRs for topics that have no receivers, but at the expense of introducing both TQRs and TIRs.)
- In a deployment that includes the DRO, some number of TQRs are necessary to inform the Router that the context is interested in the topic. To avoid deafness issues, it is recommended to extend the Querying Sustaining Phase, although at a reduced rate.
- If the **resolution_no_source_notification_threshold (receiver)** feature is used, some number of TQRs are necessary.

8.3 Message Batching

This section is about source-side batching - publishers batching messages when sending. See also Receive-Side Batching.

Batching many small messages into fewer network packets decreases the per-message CPU load, thereby increasing throughput. Let's say it costs 2 microseconds of CPU to fully process a message. If you process 10 messages per second, you won't notice the load. If you process half a million messages per second, you saturate the CPU. So to achieve high message rates, you have to reduce the per-message CPU cost with some form of message batching. These per-message costs apply to both the sender and the receiver.

Many people are under the impression that while batching reduces CPU load, it increases message latency. There are circumstances where this can happen, but it is also true that careful use of batching can result in small latency increases or none at all. In fact, there are common circumstances where Intelligent Batching will reduce average latency.

The most common forms of source-side batching employed with UM are:

- Implicit Batching
- Intelligent Batching
- Application Batching

Note that implicit and intelligent batching are not supported for UMQ Brokered Queuing, Smart Sources, or Transport LBT-SMX.

When using a kernel-bypass network driver (e.g. Solarflare's Onload), there are special considerations when batching. See Datagram Max Size and Network MTU.

8.3.1 Implicit Batching

With implicit batching, UM automatically batches smaller messages into Transport Session datagrams. The implicit batching configuration options, **implicit_batching_interval (source)** (default = 200 milliseconds) and **implicit_↩ batching_minimum_length (source)** (default = 2048 bytes) govern UM implicit message batching. Although these are source options, they actually apply to the Transport Session to which the source was assigned.

See **Implicit Batching Options**.

See also Source Configuration and Transport Sessions.

UM establishes the implicit batching parameters when it creates the Transport Session. Any sources assigned to that Transport Session use the implicit batching limits set for that Transport Session, and the limits apply to any and all sources subsequently assigned to that Transport Session. This means that batched transport datagrams can contain messages on multiple topics.

Implicit Batching Operation

Implicit Batching buffers messages until:

- the buffer size exceeds the configured **implicit_batching_minimum_length (source)**, or
- the oldest message in the buffer has been in the buffer for **implicit_batching_interval (source)** milliseconds, or
- adding another message would cause the buffer to exceed the configured Datagram Max Sizes for the underlying transport type.

When at least one condition is met, UM flushes the buffer, pushing the messages onto the network.

Note that the two size-related parameters operate somewhat differently. When the application sends a message, the **implicit_batching_minimum_length (source)** option will trigger a flush *after* the message is sent. I.e. a sent datagram will typically be larger than the value specified by **implicit_batching_minimum_length (source)** (hence the use of the word "minimum"). In contrast, the transport_∗_datagram_max_size option will trigger a flush *before* the message is sent. I.e. a sent datagram will never be larger than the transport_∗_datagram_max_size option. If both size conditions apply, the Datagram Max Sizes takes priority.

It may appear this design introduces significant latencies for low-rate topics. However, remember that Implicit Batching operates on a Transport Session basis. Typically many low-rate topics map to the same Transport Session, providing a high aggregate rate. The **implicit_batching_interval (source)** option is a last resort to prevent messages from becoming stuck in the Implicit Batching buffer. If your UM deployment frequently uses the **implicit_batching↩ _interval (source)** to push out the data (i.e. if the entire Transport Session has periods of inactivity longer than the value of **implicit_batching_interval (source)** (defaults to 200 ms), then either the implicit batching options need to be fine-tuned (reducing one or both), or you should consider an alternate form of batching. See Intelligent Batching.

The minimum value for the **implicit_batching_interval (source)** is 3 milliseconds. The actual minimum amount of time that data stays in the buffer depends on your Operating System and its scheduling clock interval. For example, on a Solaris 8 machine, the actual time is can be as much as 20 milliseconds. On older Microsoft Windows machines, the time can be as much as 16 milliseconds. On a Linux 2.6 kernel, the actual time is 3 milliseconds (+/-1).

Implicit Batching Example

The following example demonstrates how the **implicit_batching_minimum_length (source)** is actually a trigger or floor, for sending batched messages. It is sometimes misconstrued as a ceiling or upper limit.

```
source implicit_batching_minimum_length 2000
```

1. The first send by your application puts 1900 bytes into the batching buffer, which is below the minimum, so UM holds it.
2. The second send fills the batching buffer to 3800 bytes, well over the minimum. UM sends it down to the transport layer, which builds a 3800-byte (plus overhead) datagram and sends it.
3. The sender's Operating System performs IP fragmentation on the datagram to produce packets, and the receiving Operating System reassembles the datagram.
4. UM reads the datagram from the socket at the receiver.
5. UM parses out the two messages and delivers them to the appropriate topic levels, which deliver the data.

The proper setting of the implicit batching parameters often represents a trade-off between latency and efficiency, where efficiency affects the highest throughput attainable. In general, a large minimum length setting increases efficiency and allows a higher peak message rate, but at low message rates a large minimum length can increase latency. A small minimum length can lower latency at low message rates, but does not allow the message rate to reach the same peak levels due to inefficiency. An intelligent use of implicit batching and application-level flushing can be used to implement an adaptive form of batching known as Intelligent Batching which can provide low latency and high throughput with a single setting.

8.3.2 Intelligent Batching

Intelligent Batching uses Implicit Batching along with your application's knowledge of the messages it must send. It is a form of dynamic adaptive batching that automatically adjusts for different message rates. Intelligent Batching can provide significant savings of CPU resources without adding any noticeable latency.

For example, your application might receive input events in a batch, and therefore know that it must produce a corresponding batch of output messages. Or the message producer works off of an input queue, and it can detect messages in the queue. In any case, if the application knows that it has more messages to send without going to sleep, it simply does normal sends to UM, letting Implicit Batching send only when the buffer meets the **implicit_batching_minimum_length (source)** threshold.

However, when the application detects that it has no more messages to send after it sends the current message, it sets the FLUSH flag (LBM_MSG_FLUSH) when sending the message which instructs UM to flush the implicit batching buffer immediately by sending all messages to the transport layer. Refer to **lbm_src_send()** in the UM API documentation (UM C API, UM Java API, or UM .NET API) for all the available send flags.

When using Intelligent Batching, it is usually advisable to increase the **implicit_batching_minimum_length (source)** option to 10 times the size of the average message, to a maximum value of 8196. This tends to strike a good balance between batching length and flushing frequency, giving you low latencies across a wide variation of message rates.

8.3.3 Application Batching

In all of the above situations, your application sends individual messages to UM and lets UM decide when to push the data onto the wire (often with application help). With application batching, your application buffers messages itself and sends a group of messages to UM with a single send. Thus, UM treats the send as a single message. On the receiving side, your application needs to know how to dissect the UM message into individual application messages.

This approach is most useful for Java or .NET applications where there is a higher per-message cost in delivering an UM message to the application. It can also be helpful when using an event queue to deliver received messages. This imposes a thread switch cost for each UM message. At low message rates, this extra overhead is not noticeable. However, at high message rates, application batching can significantly reduce CPU overhead.

8.3.4 Explicit Batching

Warning

> **The Explicit Batching feature is deprecated and may be removed in a future release**. Users are advised to use Implicit Batching or Intelligent Batching.

UM allows you to group messages for a particular topic with explicit batching. The purpose of grouping messages with explicit batching is to allow the receiving application to detect the first and last messages of a group without needing to examine the message contents.

Note

> Explicit Batching does not guarantee that all the messages of a group will be sent in a single datagram.

Warning

> Explicit Batching does not provide any kind of transactional guarantee. It is possible to receive some messages of a group while others are unrecoverably lost. If the first and/or last messages of a group are unrecoverably lost, then the receiving application will not have an indication of start and/or end of the group.

When your application sends a message (**lbm_src_send()**) it may flag the message as being the start of a batch (LBM_MSG_START_BATCH) or the end of a batch (LBM_MSG_END_BATCH). All messages sent between the start and end are grouped together. The flag used to indicate the end of a batch also signals UM to send the message immediately to the implicit batching buffer. At this point, Implicit Batching completes the batching operation. UM includes the start and end flags in the message so receivers can process the batched messages effectively.

Unlike Intelligent Batching which allows intermediate messages to trigger flushing according to the **implicit_batching_minimum_length (source)** option, explicit batching holds all messages until the batch is completed. This feature is useful if you configure a relatively small **implicit_batching_minimum_length (source)** and your application has a batch of messages to send that exceeds the **implicit_batching_minimum_length (source)**. By releasing all the messages at once, Implicit Batching maximizes the size of the network datagrams.

Explicit Batching Example

The following example demonstrates explicit batching.

```
source implicit_batching_minimum_length 8000
```

1. Your application performs 10 sends of 100 bytes each as a single explicit batch.
2. At the 10th send (which completes the batch), UM delivers the 1000 bytes of messages to the implicit batch buffer.
3. Let's assume that the buffer already has 7899 bytes of data in it from other topics on the same Transport Session
4. UM adds the first 100-byte message to the buffer, bringing it to 7999.

5. UM adds the second 100-byte message, bringing it up to 8099 bytes, which exceeds **implicit_batching_↩ minimum_length (source)** but is below the 8192 Datagram Max Sizes.

6. UM sends the 8099 bytes (plus overhead) datagram.

7. UM adds the third through tenth messages to the implicit batch buffer. These messages will be sent when either **implicit_batching_minimum_length (source)** is again exceeded, or the **implicit_batching_interval (source)** is met, or a message arrives in the buffer with the flush flag (LBM_MSG_FLUSH) set.

8.3.5 Adaptive Batching

Warning

> **The Adaptive Batching feature is deprecated and may be removed in a future release**. Users are advised to use Implicit Batching or Intelligent Batching.

8.4 Message Fragmentation and Reassembly

Message fragmentation is the process by which an arbitrarily large message is split into a series of smaller pieces or *fragments*. Reassembly is the process of putting the pieces back together into a single contiguous message. Ultra Messaging performs UM fragmentation and reassembly on large user messages. When a user message is small enough, it fits into a single fragment.

Note that there is another layer of fragmentation and reassembly that happens in the TCP/IP network stack, usually by the host operating system. This IP fragmentation of datagrams into packets happens when sending datagrams larger than the MTU of the network medium, usually 1500 bytes. However, this fragmentation and reassembly happens transparently to and independently of Ultra Messaging. In the UM documentation, "fragmentation" generally refers to the higher-level UM fragmentation.

Some users of UM have configured their network and systems for "jumbo frames", which provides an MTU of up to 9000 bytes. This can provide efficiency benefits, but is not practical for most users. This document assumes an MTU of 1500.

Another term that Ultra Messaging borrows from networking is "datagram". In the UM documentation, a *datagram* is a unit of data which is sent to the transport (network socket or shared memory). In the case of network-based transport types, this refers to a buffer which is sent to the network socket in a single system call.

(Be aware that for UDP-based transport types (LBT-RM and LBT-RU), the UM datagrams are in fact sent as UDP datagrams. For non-UDP-based transports, the use of the term "datagram" is retained for consistency.)

The mapping of message fragments to datagrams depends on three factors:

1. User message size,

2. Configured Datagram Max Sizes for the source's transport type, and

3. Use of the Implicit Batching feature.

When configured, the source implicit batching feature combines multiple small user messages into a single datagram no greater than the size of the transport type's configured Datagram Max Sizes.

Large user messages can be split into N fragments, the first N-1 of which are approximately the size of the transport type's configured Datagram Max Sizes, and the Nth fragment containing the left-over bytes.

A subscriber's receiver callback can get information about a message's fragments by calling **lbm_msg_retrieve_fragment_info()**.

A publisher can test an individual sent message to see if UM fragmented it via the **LBM_SRC_EVENT_SEQUENCE_NUMBER_INFO** source event, which is enabled by sending the message using the **lbm_src_send_ex()** function and setting the **LBM_SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO** flag or **LBM_SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO_FRAGONLY** flag.

Note

> The SMX transport type does not support fragmentation.

8.4.1 Datagram Max Sizes

Each transport type has its own default maximum datagram size. For example, LBT-RM and LBT-RU have 8K as their default maximum datagram sizes, while TCP and IPC have 64K as their default maximums. These different defaults represent optimal values for the different transport types, and in a simple UM network, it is usually not necessary to change them.

The configuration options are:

- **transport_tcp_datagram_max_size (context)**
- **transport_lbtrm_datagram_max_size (context)**
- **transport_lbtru_datagram_max_size (context)**
- **transport_lbtipc_datagram_max_size (context)**
- **transport_lbtsmx_datagram_max_size (source)** - note this is a source-scoped option; the others are context-scoped.

Note that the transport's datagram max size option limits the size of the UM *payload*, and does not include overhead specific to the underlying transport type. For example, **transport_lbtrm_datagram_max_size (context)** does not include the UDP, IP, or packet overhead. The actual network frame can be larger than than the configured datagram max size.

Attention

> When a DRO is being used in a UM network, it is usually advisable to override the different transports' datagram max sizes to make them all the same. This should be done for all applications and UM daemons, in all Topic Resolution Domains (TRDs). See DRO **Protocol Conversion**.

8.4.2 Datagram Max Size and Network MTU

By default, when a UM source is building an outgoing message, it reserves space for the largest possible UM header. This fixed-size reservation is subtracted from the configured datagram max size to determine how much user data can be added to the datagram. Since most UM messages do not need a large UM header, there can be cases where a user message that *could* fit in a single datagram without exceeding the configured datagram limit will instead be split into two fragments, each carried in a datagram that is significantly below the configured datagram limit.

When UM is configured with a datagram max size of 8K or more, this space inefficiency does not pose a noticeable problem since the header reservation is a small percentage of the full 8K. However, for users who want to avoid

IP fragmentation, the datagrams should be limited to 1472 bytes. In this case, the space inefficiency becomes objectionable; packets will often not be as full as they should be.

For most networks, Informatica recommends setting the Datagram Max Sizes to a minimum of 8K, and allowing the operating system to perform IP fragmentation. Some network engineers have learned that IP fragmentation can place a burden on network hardware (routers and switches), but this is true only if the network network hardware has to perform the fragmentation. For most modern networks, the entire fabric is designed to handle a common MTU, typically of 1500 bytes. Thus, a UDP datagram larger than 1500 bytes is fragmented once by the sending host's operating system, and the switches and routers only need to forward the already-fragmented packets. Switches and routers can forward fragmented packets without burden.

There are typically two circumstances where users should set UM's Datagram Max Sizes to a smaller value:

- If kernel bypass network drivers are being used which either don't handle IP fragments at all, or does handle them but only via the "slow path" (i.e. not accelerated).
- If the smallest MTU across a network path is smaller than the MTU of a host's network interface. For example, your hosts typically have an Ethernet connection with a 1500-byte MTU. But a WAN link might have a much smaller MTU. In that case, UM's Datagram Max Sizes should be set to a value that keeps all packets within the WAN link's MTU. Note that in modern networks, it is rare to find links with an MTU below 1500.

In both cases, the space inefficiency caused by reserving space for the largest possible UM header is objectionable.

The preferred solution is to use UM version 6.14 or beyond and enable the Dynamic Fragmentation Reduction feature.

For pre-6.14 UM, it can be possible to artificially increase the Datagram Max Sizes to a value above the desired limit. See Setting Datagram Max Sizes High.

Note

> UM version 6.12 changed the amount of space that Smart Sources reserve for the UM header. This can mean that pre-6.12 smart source users upgrading to 6.12 and beyond may need to change their configuration. See **Smart Source Header Size Change**.

8.4.3 Setting Datagram Max Sizes High

There are cases when the Datagram Max Sizes must be set to a relatively small value to prevent IP fragmentation. However, as described in Datagram Max Size and Network MTU, this is space inefficient and can result in more UM fragmentation than is necessary.

Some users set their datagram max size to a value *above* the desired limit, knowing that LBT-RM and LBT-RU will reserve a large number of bytes for the largest possible UM header. This allows traffic to more-efficiently fill packets, and therefore avoid unnecessary UM fragmentation. However, this technique is imperfect.

Setting the datagram max size above the desired limit technically gives UM permission to send a datagram up to that size. I.e. you cannot always count on the largest possible header space being reserved. In practice with a restricted use case (e.g. flushing every message), LBT-RM data messages may never go above the desired limit. But TSNI messages can reach the full datagram max size, and so can implicitly batched messages.

Also, keep in mind that UM does not publish the internal reserved size, and does not guarantee that the reserved size will stay the same. Users who use this technique must determine their optimal datagram max size *empirically* through extensive testing within the constraints of their use cases.

As of UM version 6.14, Informatica recommends that kernel bypass users make use of the Dynamic Fragmentation Reduction feature. This allows setting the datagram max size to 1472 to prevent IP fragmentation, while ensuring that data messages are not unnecessarily fragmented.

8.4.4 Dynamic Fragmentation Reduction

As described in Datagram Max Size and Network MTU, UM versions prior to 6.14 reserved an unnecessarily large size in each datagram for a worst-case header, resulting in the LBT-RM and LBT-RU protocols performing UM fragmentation at message sizes where they should not have to. In an effort to more-fully fill data packets, some users set their Datagram Max Sizes above an MTU, but this can result in IP fragmentation.

As of UM version 6.14, enabling **dynamic_fragmentation_reduction (context)** lets UM's transport protocols more-fully utilize the configured datagram max size, thus avoiding unnecessary UM and IP fragmentation. This option is typically only of interest to users of LBT-RM and/or LBT-RU who need to avoid IP fragmentation, such as users of a kernel-bypass driver.

This allows the user to set the desired transport's datagram max size option to 1472, which prevents IP fragmentation on the transport session, while still efficiently filling the packets to close to the MTU.

Enabling Dynamic Fragmentation Reduction

Users of kernel bypass drivers sometimes set their datagram max sizes well above 1472. This avoids inefficient filling of packets, but still carries the risk that some messages will trigger IP fragmentation.

To use the Dynamic Fragmentation Reduction feature:

```
context dynamic_fragmentation_reduction 1
context transport_lbtrm_datagram_max_size 1472
context transport_lbtru_datagram_max_size 1472
# Technically, only RM and RU matter. But when DROs are present, allow
# protocol conversion by making all transports the same.
context transport_tcp_datagram_max_size 1472
context transport_lbtipc_datagram_max_size 1472
source transport_lbtsmx_datagram_max_size 1472
```

DRO users see **Protocol Conversion**.

Upgrade Path

The Dynamic Fragmentation Reduction feature is designed to allow a gradual upgrade. Older versions of UM can interoperate with 6.14 and beyond using Dynamic Fragmentation Reduction, but certain requirements must be met.

Users interested in Dynamic Fragmentation Reduction are typically users of kernel-bypass drivers who want to set their datagram max sizes to 1 MTU. For efficiency purposes, they've empirically determined an optimal value which is noticeably higher than a network MTU. A typical value might be between 1800 and 1900.

During the upgrade period, you will be running upgraded UM programs (version 6.14 or beyond) with Dynamic Fragmentation Reduction enabled, and datagram max sizes set to 1472. These will successfully interoperate with older UM versions running with datagram max sizes between 1800 and 1900.

The exception to this rule is the DRO. In a mixed-version environment, the DRO should always be configured with the largest datagram max size values used in your network. I.e. if an older version application is set to 1800, a DRO running 6.14 or beyond should also be configured for 1800.

See also **Protocol Conversion**.

8.5 Ordered Delivery

With the Ordered Delivery feature, a receiver's Delivery Controller can deliver messages to your application in sequence number order or arrival order. This feature can also reassemble fragmented messages or leave reassembly to the application. You can set Ordered Delivery via UM configuration option to one of three modes:

- Sequence Number Order, Fragments Reassembled
- Arrival Order, Fragments Reassembled

- Arrival Order, Fragments Not Reassembled (deprecated)

See **ordered_delivery (receiver)**

Note that these ordering modes only apply to a specific topic from a single publisher. UM does not ensure ordering across different topics, or on a single topic across different publishers. See Message Ordering for more information.

8.5.1 Sequence Number Order, Fragments Reassembled (Default Mode)

In this mode, a receiver's Delivery Controller delivers messages in sequence number order (the same order in which they are sent). This feature also guarantees reassembly of fragmented large messages. To enable sequence number ordered delivery, set the **ordered_delivery (receiver)** configuration option as shown:

```
receiver ordered_delivery 1
```

Please note that ordered delivery can introduce latency when packets are lost (new messages are buffered waiting for retransmission of lost packets).

8.5.2 Arrival Order, Fragments Reassembled

This mode delivers messages immediately upon reception, in the order the datagrams are received, except for fragmented messages, which UM holds and reassembles before delivering to your application. Be aware that messages can be delivered out of order, either because of message loss and retransmission, or because the networking hardware re-orders UDP packets. Your application can then use the sequence_number field of **lbm_msg_t** objects to order or discard messages. But be aware that the sequence number may not always increase by 1; application messages larger than the Datagram Max Sizes will be split into fragments, and each fragment gets its own sequence number. With the "Arrival Order, Fragments Reassembled" mode of delivery, UM will reassemble the fragments into the original large application message and deliver it with a single call to the application receiver callback. But that message's sequence_number will reflect the final fragment.

To enable this arrival-order-with-reassembly mode, set the following configuration option as shown:

```
receiver ordered_delivery -1
```

8.5.3 Arrival Order, Fragments Not Reassembled

Warning

> This mode of delivery is deprecated and may be removed in a future version. The user is advised to use mode -1.

This mode allows messages to be delivered to the application immediately upon reception, in the order the datagrams are received. If a message is lost, UM will retransmit the message. In the meantime, any subsequent messages received are delivered immediately to the application, followed by the dropped packet when its retransmission is received. This mode has the lowest latency.

With this mode, the receiver delivers messages larger than the transport's Datagram Max Sizes as individual fragments. The C API function, **lbm_msg_retrieve_fragment_info()** returns fragmentation information for the message you pass to it, and can be used to reassemble large messages. (In Java and .NET, LBMMessage provides methods to return the same fragment information.) Note that reassembly is not required for small messages.

To enable this no-reassemble arrival-order mode, set the following configuration option as shown:

```
receiver ordered_delivery 0
```

When developing message reassembly code, consider the following:

- Message fragments don't necessarily arrive in sequence number order.
- Some message fragments may never arrive (unrecoverable loss), so you must time out partial messages.

Arrival order delivery without reassembly is not compatible with the following UM features:

- Transport LBT-SMX
- Message Properties

8.6 Loss Detection Using TSNIs

A TSNI (Topic Sequence Number Information) message is an internal UM message sent by a source to inform receivers of the topic-level sequence number of the last message sent.

When a source enters an idle period during which it has no data traffic to send (5 seconds by default), that source will send one or more TSNI messages. The TSNI lets receivers know that the source is still active and also reminds receivers of the sequence number of the last message. This helps receivers become aware of any lost messages prior to the TSNI.

Sources send TSNIs over the same transport and on the same topic as normal data messages. You can set a time value of the TSNI interval with configuration option **transport_topic_sequence_number_info_interval (source)**. You can also set a time value for the duration that the source sends contiguous TSNIs with configuration option **transport_topic_sequence_number_info_active_threshold (source)**, after which time the source stops issuing TSNIs.

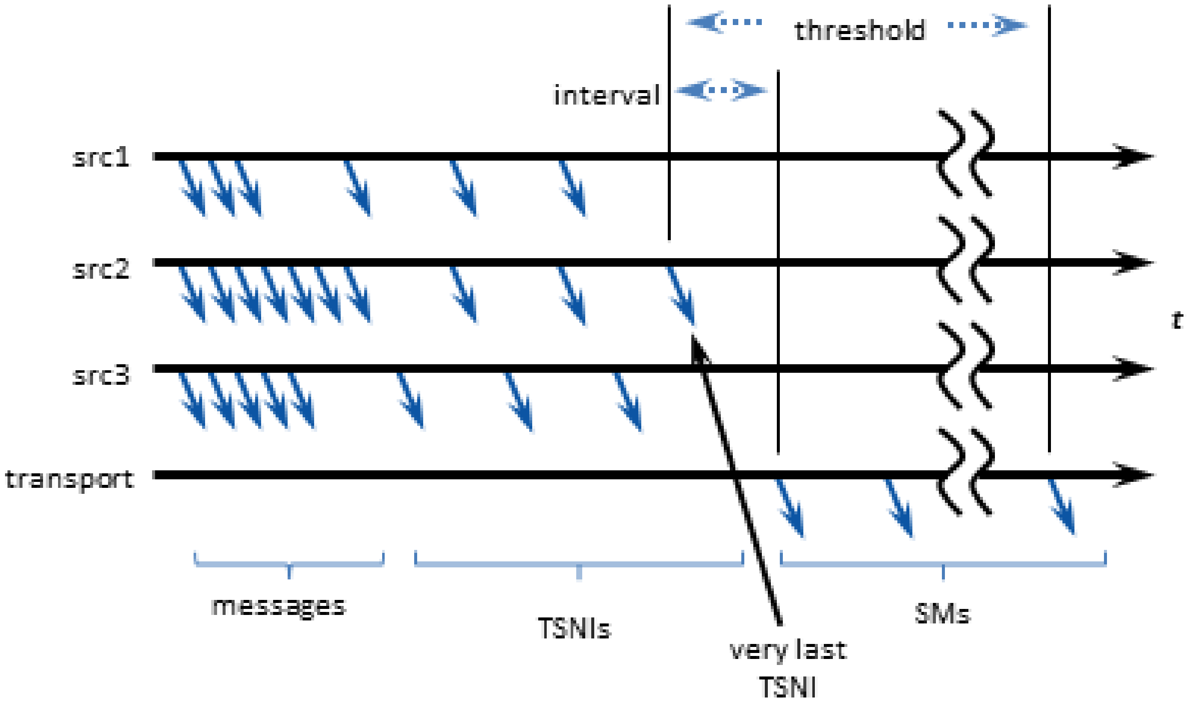
8.7 Receiver Keepalive Using Session Messages

When an LBT-RM, LBT-RU, or LBT-IPC Transport Session enters an inactive period during which it has no messages to send, the UM context sends Session Messages (SMs). The first SM is sent after 200 milliseconds of inactivity (by default). If the period of inactivity continues additional SMs will be sent at increasing intervals, up to a maximum interval of 10 seconds (by default).

SMs serve three functions:

1. **Keepalive** - SMs inform receivers that transport sessions are still alive. If a receiver stops getting any kind of traffic for a transport session, after a configurable period of inactivity the receiver will time out the transport session and will assume that it has died.
2. **Tail loss** - for UDP-based transport sessions (LBT-RM and LBT-RU), SMs are used to detect packet loss, specifically "tail loss", and trigger recovery.
3. **Multicast Flows** - for multicast-based transport sessions (LBT-RM), SMs serve to keep the network hardware multicast flows "hot", so that replication and forwarding of multicast packets is done in hardware at line speed.

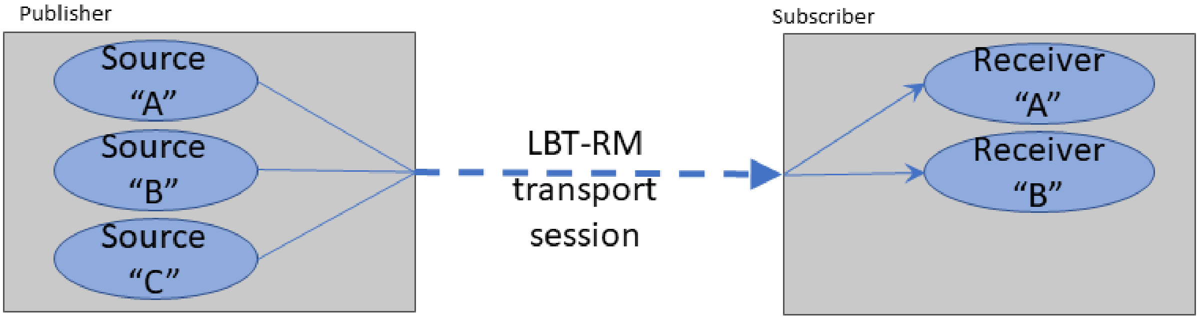
Any other UM message on a transport session will suppress the sending of SMs, including data messages and TSNIs. (Topic Resolution messages are not sent on the transport session, and will not suppress sending SMs.) You can set time values for SM interval and duration with configuration options specific to their transport type.



8.8 Extended Messaging Example

This section illustrates many of the preceding concepts using an extended example of message passing. This example uses LBT-RM, but for the purposes of this example, LBT-RU operates in a similar manner.

The example starts out with two applications, Publisher and Subscriber:



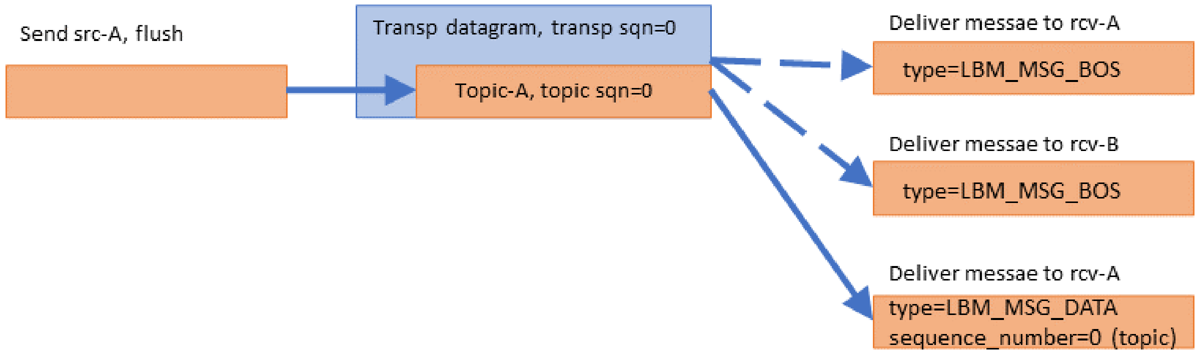
The publisher has created three source objects, for topics "A", "B", and "C" respectively. All three sources are mapped to a single LBT-RM Transport Session by configuring them for the same **multicast group address** and **destination port**.

The Subscriber application creates two receivers, for topics "A" and "B".

The creation of sources and receivers triggers Topic Resolution, and the subscriber joins the Transport Session once the topics are resolved. To be precise, the first receiver to discover a source triggers joining the Transport Session and creating a Delivery Controller; subsequent source discoveries on the same Transport Session don't need to join; they only create Delivery Controllers. However, until such time as one or more publishing sources send their first topic-layer message, the source Transport Session sends no datagrams. The Transport Session is created, but has not yet "started".

8.8.1 Example: First Message

In this example, the first message on the Transport Session is generated by the publishing application sending an application message, in this case for topic "A".



The send function is passed the "flush" flag so that the message is sent immediately. The message is assigned a topic-level sequence number of 0, since it is the application's first message for that topic. The source-side transport layer wraps the application message in a datagram and gives it transport sequence number 0, since it is the first datagram sent on the Transport Session.

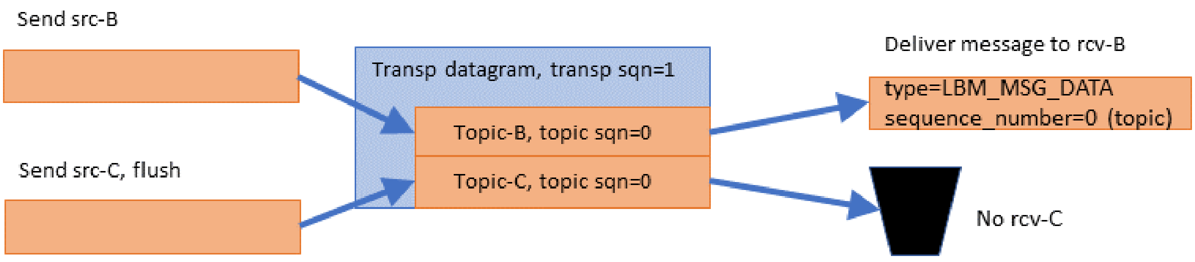
On the receive side, the first datagram (of any kind) on the Transport Session informs the transport layer that the Transport Session is active. The transport layer informs all mapped Delivery Controller instances that the Transport Session has begun. Each Delivery Controller delivers a Beginning Of Session event (BOS) to the application callback for each receiver. The passed-in **lbm_msg_t** structure has event **type** equal to **LBM_MSG_BOS**.

Note that the receiver for topic B gets a BOS even though no messages were received for it; the BOS event informs the receivers that the *Transport Session* is active, not the topic.

Finally, the transport layer passes the received datagram to the topic-A Delivery Controller, which passes the application message to the receiver callback. The passed-in **lbm_msg_t** structure has event **type** equal to **LBM←_MSG_DATA**, and a topic-level **sequence_number** of 0. (The transport sequence number is not available to the application.)

8.8.2 Example: Batching

The publishing application now has two more messages to send. To maximize efficiency, it chooses to batch the messages together:



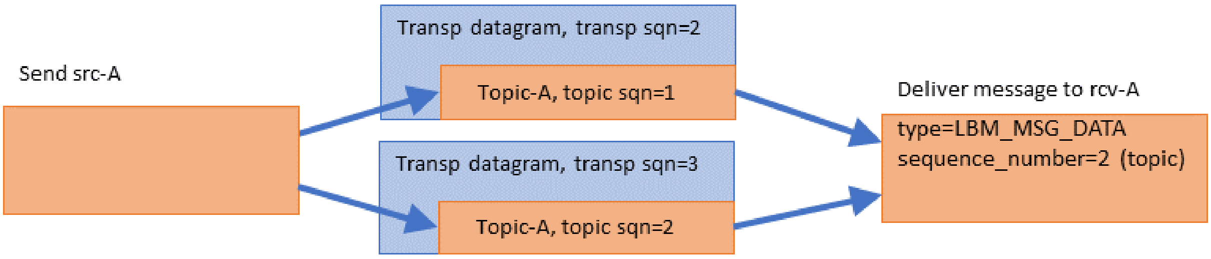
The publishing application sends a message to topic "B", this time *without* the "flush" flag. The source-side topic layer buffers the message. Then the publishing application sends a message to topic "C", *with* the "flush" flag. The source-side transport layer wraps both application messages into a single datagram and gives it transport sequence number 1, since it is the second datagram sent on the Transport Session. But the two topic level sequence numbers are 0, since these are the first messages sent to those topics.

Note that almost no latency is added by batching, so long as the second message is ready to send immediately after the first. This method of low-latency batching is called Intelligent Batching, and can greatly increase the maximum sustainable throughput of UM.

The subscriber gets the datagram and delivers the topic "B" message to the application receiver callback. It's topic-level **sequence_number** is 0 since it was the first message sent to the "B" source. However, the subscriber application has no receiver for topic "C", so the message "C" is simply discarded.

8.8.3 Example: UM Fragmentation

The publishing application now has a topic "A" message to send that is larger than the **maximum allowable datagram**.



The source-side topic layer splits the application message into two fragments and assigns each fragment its own topic-level sequence number (1 for the first, 2 for the second). The topic-layer gives each fragment separately to the transport layer, which wraps each fragment into its own datagram, consuming two transport sequence numbers (2 and 3). Note that the transport layer does not interpret these fragments as parts of a single larger message; from the transport's point of view, this simply two datagrams being sent.

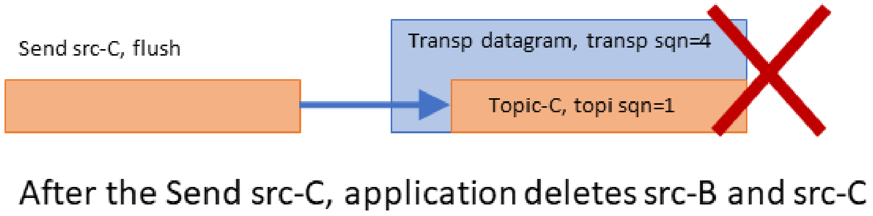
The receive-side transport layer gets the datagrams and hands them to the Topic-A Delivery Controller (receiver-side topic layer). The Delivery Controller reassembles the fragments in the correct order, and delivers the message to the application's receiver callback in a single call. The **sequence_number** visible to the application is the topic-level sequence number of the *last* fragment (2 in this example).

Note that the application receiver callback never sees a topic sequence_number of 1 for topic "A". It saw 0 then 2, with 1 seemingly missing. However, the application can call **lbm_msg_retrieve_fragment_info()** to find out the range of topic sequence numbers consumed by a message.

The behavior described above is for the default **ordered_delivery (receiver)** equal to 1. see Ordered Delivery for alternative behaviors.

8.8.4 Example: Loss Recovery

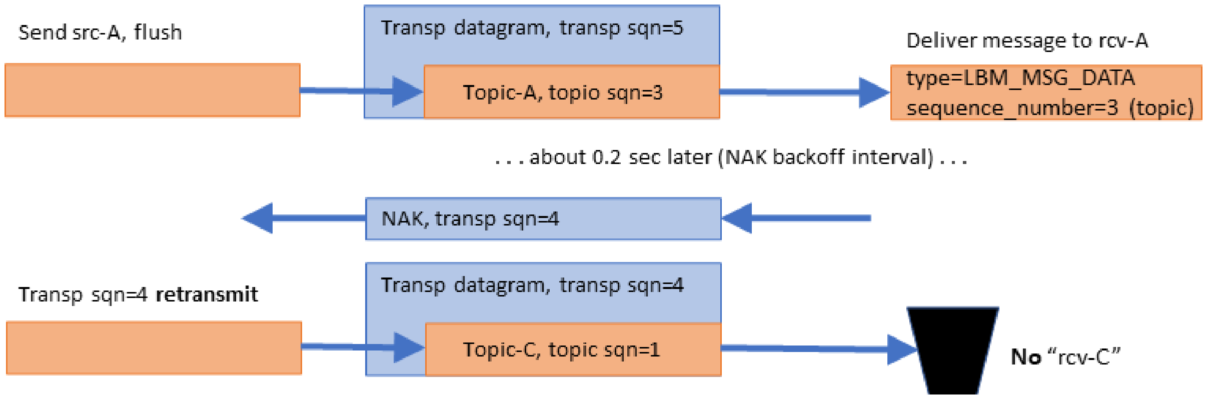
Now the publishing application sends a message to topic C. But the datagram is lost, so the receiver does not see it. Also, right after the send to topic C, the application deletes the sources for topics B and C.


Send src-C, flush
Transp datagram, transp sqn=4
Topic-C, topi sqn=1
After the Send src-C, application deletes src-B and src-C

Deleting a source shortly after sending a message to it is contrary to best practice. Applications should pause between the last send to a topic and the deletion of the topic, preferable a delay of between 5 and 10 seconds. This gives receivers an opportunity to attempt recovery if the last message sent was lost. We delete the sources here to illustrate an important point.

Note that although the datagram was lost and two topics were deleted, nothing happens. The receiver does not request a retransmission because the receiver has no idea that the source sent a message. Also, the source-side topic layer does not explicitly inform the receiver that the topics are deleted.

Continuing the example, the publishing application sends another message, this time a message for topic A ("Topic-A, topic sqn=3"):



There are two notable events here:

1. The "A" message is delivered immediately to the topic "A" receiver, even though earlier data was lost and not yet retransmitted. If this were TCP, the kernel would buffer and prevent delivery of subsequent data until the lost data is recovered.
2. The reception of that "A" message with transport sequence number 5 informs the receive-side transport layer that transport datagram #4 was lost. So it initiates a NAK/retransmission cycle. When the lost datagram is retransmitted, the receiver throws it away since it is for an unsubscribed topic.

You might wonder: why NAK and retransmit datagram 4 if the subscriber is just going to throw it away? The subscriber NAKs it because it has no way of knowing which topic it contains; if it were topic B, then it would need that datagram. The publisher retransmits it because it does not know which topics the subscriber is interested in. It has no way of knowing that the subscriber will throw it away.

Regarding message "Topic-A, sqn=3", what if the publisher did not have that message to send? I.e. what if that "Topic-C, sqn=1" message were the last one for a while? This is called "tail loss" since the lost datagram is not immediately followed by a successful datagram. The subscriber has no idea that messages were sent but lost. In this case, the source-side transport layer would have sent a transport-level "session message" after about 200 ms of inactivity on the Transport Session. That session message would inform the receiver-side transport layer that datagram #5 was lost, and would trigger the NAK/retransmission.

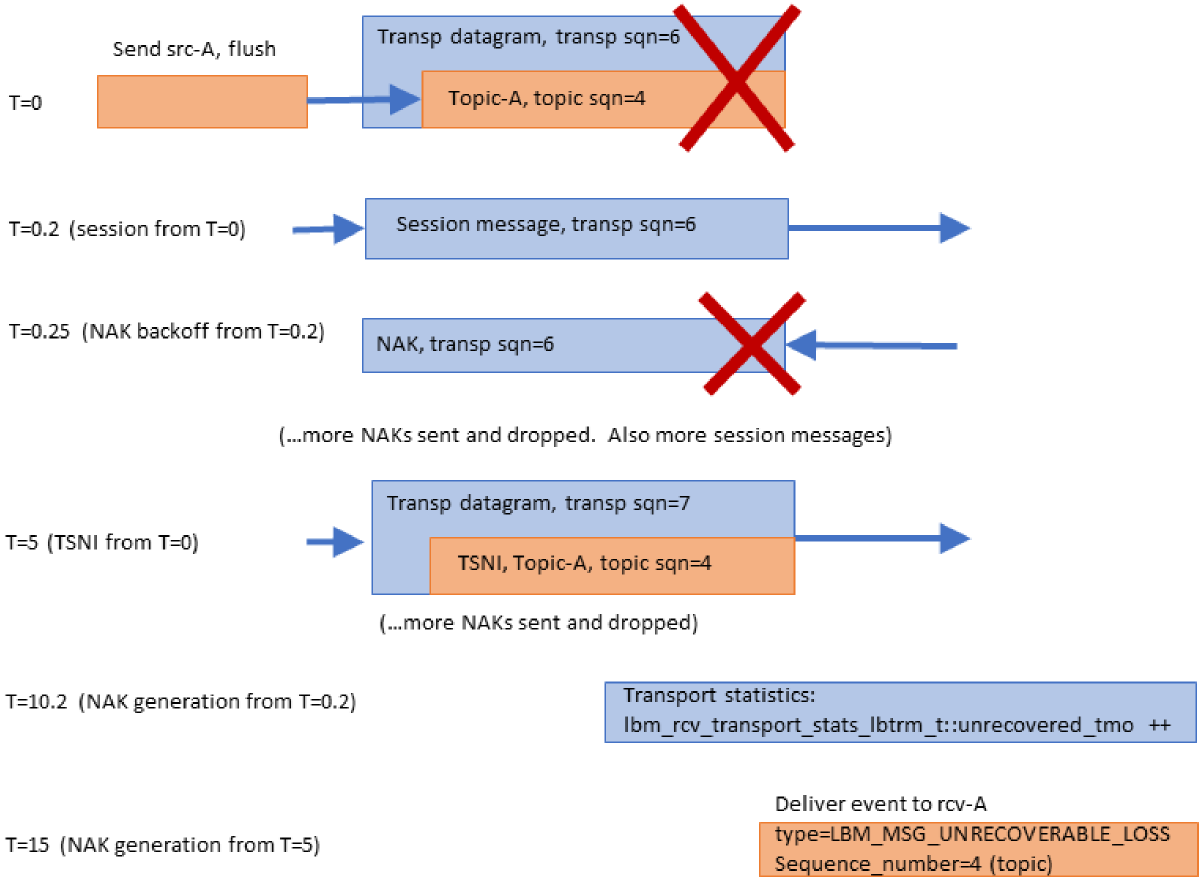
Finally, note that the message for topic-C was retransmitted, even though the topic-C source was deleted. This is because the deletion of a source does not purge the transport layer's retransmission buffer of datagrams from that source. However, higher-level recovery mechanisms, such as late join and OTR, are no longer possible after the source is deleted. Also, if *all* sources on a Transport Session are deleted, the Transport Session itself is deleted, which makes even transport-level retransmission impossible. (Only Persistence allows recovery after the transport session is deleted.)

8.8.5 Example: Unrecoverable Loss

The previous examples assume that events are happening in fairly rapid succession. In this example of unrecoverable loss, significantly longer time periods are involved.

Unrecoverable loss is what happens when UM tries to recover the lost data but it is unable to. There are many possible scenarios which can cause recovery efforts fail, most of which involve a massive overload of one or more components in the data flow.

To simplify this example, let's assume that, starting now, all NAKs are blocked by the network switch. If the publisher never sees the NAKs, it assumes that all datagrams were received successfully and does not retransmit anything.


Send src-A, flush
T=0
Transp datagram, transp sqn=6
Topic-A, topic sqn=4
T=0.2 (session from T=0)
Session message, transp sqn=6
T=0.25 (NAK backoff from T=0.2)
NAK, transp sqn=6
(...more NAKs sent and dropped. Also more session messages)
T=5 (TSNI from T=0)
Transp datagram, transp sqn=7
TSNI, Topic-A, topic sqn=4
(...more NAKs sent and dropped)
T=10.2 (NAK generation from T=0.2)
Transport statistics:
lbm_rcv_transport_stats_lbtrm_t::unrecovered_tmo ++
Deliver event to rcv-A
type=LBM_MSG_UNRECOVERABLE_LOSS
Sequence_number=4 (topic)
T=15 (NAK generation from T=5)

At T=0, the message "Topic-A, sqn=4" is sent, but not received. Let's assume that the publisher has no more application messages to send for a while. With every application message sent, the source starts two activity timers: a transport-level "session" timer, and a topic-level "TSNI" timer. The session timer is for .2 seconds (see

transport_lbtrm_sm_minimum_interval (source)), and the TSNI timer is for 5 seconds (see **transport_topic_sequence_number_info_interval (source)**).

At T=0.2, the session timer expires and the source-side transport layer sends a session message. When the receive-side transport layer sees the session message, it learns that transport datagram #6 was lost. So it starts two receive-side transport-level timers: "NAK backoff" and "NAK generation". NAK backoff is shown here as .05 seconds, but is actually randomized between .025 and .075 (see **transport_lbtrm_nak_initial_backoff_interval (receiver)**. The NAK generation is 10 seconds (see **transport_lbtrm_nak_generation_interval (receiver)**).

At T=0.25, the NAK backoff timer expires. Since the transport receiver still has not seen datagram #6, it sends a NAK. However, we are assuming that all NAKs are blocked, so the transport source never sees it. Over the next ∼5 seconds, the source will send several more session messages and the receiver will send several more NAKs (not shown).

At T=5, the TSNI timer set by the source at T=0 expires. Since no application messages have been sent since then, the source sends a TSNI message for topic "A". This informs the Delivery Controller that it lost the message "Topic-A, sqn=4". However, the receive-side Delivery Controller (topic layer) does not initiate any recovery activity. It only sets a *topic-level* timer for the same amount of time as the transport's NAK generation timer, 10 seconds. The Delivery Controller assumes that the transport layer will do the actual data recovery.

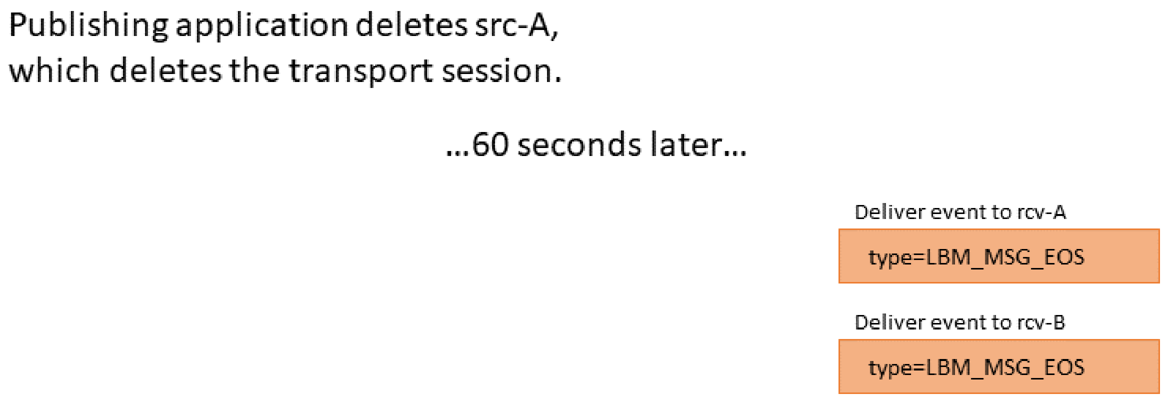
At T=10.2, the receive-side transport layer's NAK generation timer (set at T=0.2) finally expires; the *transport layer* now considers datagram #6 as unrecoverable loss. The transport layer stops sending NAKs for that datagram, and it increments the receive-side transport statistic **lbm_rcv_transport_stats_lbtrm_t_stct::unrecovered_tmo**. Note that it does *not* deliver an unrecoverable loss event to the application.

Over the next ∼5 seconds, the Delivery Controller continues to wait for the missing message to be recovered by the transport receiver, but the transport receiver has already given up. You might wonder why the transport layer doesn't inform the Delivery Controller that the lost datagram was unrecoverable loss. The problem is that the transport layer does not know the contents of the lost datagram, and therefore does not know which topic to inform. That is why the Delivery Controller needs to set its own NAK generation timer at the point where it detects topic-level loss (at T=5).

Note that had sources src-B and src-C not been deleted earlier, messages sent to them could have been successfully received and processed during this entire 15-second period. However, any subsequent messages for topic "A" would need to be buffered until T=15. After the unrecoverable loss event is delivered for topic A sequence_number 4, subsequently received and buffered messages for topic "A" are delivered.

8.8.6 Example: Transport Deletion

During the previous 15 seconds, the source-side had sent a number of topic-level TSNI (for topic A) and transport-level session messages. At this point, the publishing application deletes source "A". Since sources "B" and "C" were deleted earlier, "A" was the last source mapped to the Transport Session. So UM deletes the Transport Session.



Note that no indication is sent from the source side to inform receivers of the removal of the sources, nor the

Transport Session. So the receive-side transport layer has to time out the Transport Session after 60 seconds of inactivity (see **transport_lbtrm_activity_timeout (receiver)**).

The receive-side transport layer then informs both Delivery Controllers of the End Of Session event, which the Delivery Controllers pass onto the application receiver callback for each topic. The **lbm_msg_t** structure has an event **type** of **LBM_MSG_EOS**. The delivery controllers and the receive-side transport layer instance are then deleted.

However, note that the receiver objects will continue to exist. They are ready in case another publishing application starts up and creates sources for topics A and/or B.

Chapter 9

Application Design Principles

9.1 UM Monitoring

For an Ultra Messaging deployment, "monitoring" is the process of overseeing the operation of UM and the resources it uses to determine its health and performance.

Informatica strongly recommends that users of Ultra Messaging actively monitor its operation.

Monitoring is addressed in depth in the Operations Guide; see **Monitoring Introduction**. From an application design point of view, you should decide on a monitoring approach before you start coding your applications.

In particular, implement the recommendations related to the **Application Log File**.

9.2 Message Reception

Applications receive messages from UM via application callback. The application registers its callback function with UM during the creation of the Receiver Object. As messages are received, the application's receiver callback function is called, passing in the received message.

Note: there are events other than message reception that can trigger calls to the application's receiver callback. Those other event types are not covered in this section (see **lbm_msg_t_stct::type**).

At a high level, there are three common approaches to handling received messages:

1. Message is fully processed by the application's receiver callback, called by a context or XSP thread. This is typically the most efficient approach.

2. Message is fully processed by the application's receiver callback, called by an event queue dispatch thread. This is a simple method to move message processing to a different thread.

3. Message is retained by the receiver callback for further processing by a different application thread.

For example code demonstrating message reception best practices, see the API language:

- C Message Reception

- Java Message Reception

- .NET Message Reception

9.2.1 C Message Reception

C: Message Is Fully Processed by the Application's Receiver Callback, Called by a Context or XSP Thread

```
int my_receiver_callback(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)
{
  switch (msg->type) {
    case LBM_MSG_DATA:

      /* Process "msg->len" bytes in "msg->data". */
      ...

      break;

    /* Handle other receiver events. */
  }  /* switch */

  /* No need to delete message (C API deletes by default). */
  return 0;
}  /* my_receiver_callback */
```

Important rules regarding the receiver callback function when called by a context/XSP thread:

- The function must return 0.
- The passed-in message must not be modified.
- The function should not perform any operation that might be time consuming or put the thread to sleep (block). This is because any delays in your callback prevents the context thread from servicing its sockets, increasing the risk of packet loss.
- If the receiver callback sends a message, non-blocking sends must be used (see **LBM_SRC_NONBLOCK**). The code should be written to handle a send failure of **LBM_EWOULDBLOCK**.
- It is not allowed to create/delete sources or receivers, or subscribe/unsubscribe from a Spectrum channel from a callback function executed by a context/XSP thread.
- You may schedule UM timers.

If this is a Persistent receiver, see **Persistence Message Consumption**.

C: Message Is Fully Processed by the Application's Receiver Callback, Called by an Event Queue Dispatch Thread

The code is identical to the context/XSP thread case above.

The rules are similar, with some important differences (in bold):

- The function must return 0.
- The passed-in message must not be modified.
- **The application is permitted to perform operations that are time consuming and/or blocking, as long as the average message processing rate is greater than the average message sending rate. Any temporary delays will buffer messages in the event queue.**
- **Blocking or non-blocking sends may be used, according to the application's preferences.**
- **You may create/delete sources or receivers, or subscribe/unsubscribe from a Spectrum channel. However, the receiver which is delivering the current message must not be deleted.**
- You may schedule UM timers.

- Informatica strongly recommends monitoring event queue length. See Event Queue Monitor for more information.

If this is a Persistent receiver, see **Persistence Message Consumption**.

C: Message Is Retained by the Receiver Callback for Further Processing by a Different Application Thread

```
int my_receiver_callback(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)
{
  int err, more_processing_needed;

  switch (msg->type) {
    case LBM_MSG_DATA:
      more_processing_needed = my_process_received_message(msg);
      if (more_processing_needed) {
        err = lbm_msg_retain(msg);  /* Check and handle errors. */
        my_save_message_for_more_processing(msg);
      }
      break;

    /* Handle other receiver events. */
  }  /* switch */

  return 0;
}  /* my_receiver_callback */

...

void my_additional_processing(lbm_msg_t *msg)
{
  int err;

  /* Process "msg->len" bytes in "msg->data". */
  ...

  /* Tell UM: finished with "msg". */
  err = lbm_msg_delete(msg);
  /* Check and handle errors. */
}  /* my_additional_processing */
```

This demonstrates the use of:

- **lbm_msg_retain()**
- **lbm_msg_delete()**

By default, when a receiver callback returns, the UM message is implicitly deleted. To prevent that from happening, the message must be "retained". The application must ensure that every message retained is eventually deleted.

The my_save_message_for_more_processing() function is just whatever you use to transfer the message to your processing thread. The my_additional_processing() function is typically called by your thread to process the message.

The rules for the message processing code are the same as the event queue case (above).

If this is a Persistent receiver, see **Persistence Message Consumption**.

9.2.2 Java Message Reception

Java: Message Is Fully Processed by the Application's Receiver Callback, Called by a Context or XSP Thread

```
public int onReceive(Object cbArg, LBMMessage msg)
{
  try {
    switch (msg.type())
    {
      case LBM.MSG_DATA:
        ByteBuffer message = msg.dataBuffer();
        long msgLen = msg.dataLength();
        // Process "msgLen" bytes in "message".
        ...

        // Tell UM: finished with "msg".
        msg.dispose();
        break;

      /* Handle other receiver events. */
    }  /* switch */
  } catch (Exception e) {
    /* Handle exception. */
  }
  return 0;
}  /* onReceive */
```

This demonstrates the use of:

- **com.latencybusters.lbm.LBMMessage.dataBuffer()**
- **com.latencybusters.lbm.LBMMessage.dataLength()**
- **com.latencybusters.lbm.LBMMessage.dispose()**

These functions prevent unnecessary garbage and are strongly recommended by Informatica (see Zero Object Delivery).

Important rules regarding the receiver callback function when called by the context thread:

- The function must return 0.
- The function must not be allowed to pass an unhandled exception back into UM. For example, you could have your entire callback function enclosed in a large try/catch. (This is true of all UM callbacks, not just receiver.)
- The passed-in message must not be modified.
- The passed-in message must be disposed. In Java, every message must explicitly be disposed, to properly clean up the native memory. Do not assume that GC will clean it up.
- The function should not perform any operation that might be time consuming or put the thread to sleep (block). This is because any delays in your callback prevents the context thread from servicing its sockets, increasing the risk of packet loss.
- If the receiver callback sends a message, non-blocking sends must be used (see **com::latencybusters::lbm::LBM::SRC_NONBLOCK**). And the code should be written to handle a send failure of **com::latencybusters::lbm::LBMEWouldBlockException**.
- It is not allowed to create/delete sources or receivers, or subscribe/unsubscribe from a Spectrum channel from a callback function executed by the context thread.
- You may schedule UM timers.

Note the requirement to call `msg.dispose()` on every message. Prior to UM version 6.7, calling dispose() on every message was considered best practice, but was only *required* for certain use cases. UM version 6.7 introduced significant performance improvements with Java, but these improvements made calling `msg.dispose()` mandatory.

If this is a Persistent receiver, see **Persistence Message Consumption**.

Java: Message Is Fully Processed by the Application's Receiver Callback, Called by an Event Queue Dispatch Thread

The code is identical to the context thread case above.

The rules are similar, with some important differences (in bold):

- The function must return 0.

- The function must not be allowed to pass an unhandled exception back into UM. For example, you could have your entire callback function enclosed in a large try/catch. (This is true of all UM callbacks, not just receiver.)
- The passed-in message must not be modified.
- The passed-in message must be disposed. In Java, every message must explicitly be disposed, to properly clean up the native memory. Do not assume that GC will clean it up.
- **The application is permitted to perform operations that are time consuming and/or blocking, as long as the average message processing rate is greater than the average message sending rate. Any temporary delays will buffer messages in the event queue.**
- **Blocking or non-blocking sends may be used, according to the application's preferences.**
- **You may create/delete sources or receivers, or subscribe/unsubscribe from a Spectrum channel. However, the receiver which is delivering the current message should not be deleted.**
- You may schedule UM timers.
- Informatica strongly recommends monitoring event queue length. See Event Queue Monitor for more information.

If this is a Persistent receiver, see **Persistence Message Consumption**.

Java: Message Is Retained by the Receiver Callback for Further Processing by a Different Application Thread

```
public void init_UM()
{
  // During initialization, the UM context and receiver(s) are
  // configured to use a recycler.
  LBMObjectRecycler objRec = new LBMObjectRecycler();
...
  // Preparing to create context - set up attribute.
  LBMContextAttributes ctx_attr = new LBMContextAttributes();
  ctx_attr.setObjectRecycler(objRec, null);
  // Proceed with creating context, using ctx_attr.
...
  // Preparing to create receiver - set up attribute.
  LBMReceiverAttributes rcv_attr = new LBMReceiverAttributes();
  rcv_attr.setObjectRecycler(objRec, null);
  // Proceed with creating receiver, using rcv_attr.
...
}

...

public int onReceive(Object cbArg, LBMMessage msg)
{
  try {
    switch (msg.type())
    {
      case LBM.MSG_DATA:
        msg.promote();  // Tell UM: not finished with "msg".
        saveMessageForMoreProcessing(msg);
        break;

      /* Handle other receiver events. */
    }  /* switch */
  } catch (Exception e) {
    /* Handle exception. */
  }
  return 0;
}  /* onReceive */

...

public int myAdditonalProcessing(LBMMessage msg)
{
  ByteBuffer message = msg.dataBuffer();
  long msgLen = msg.dataLength();

  // Process "msgLen" bytes in "message".
  ...

  // Tell UM: finished with "msg".
  msg.dispose();
  objRec.doneWithMessage(msg);
}  /* myAdditonalProcessing */
```

This demonstrates the use of:

- **com.latencybusters.lbm.LBMMessage.promote()**
  - informs UM that the message is retained after the return of the receiver callback.
- **com.latencybusters.lbm.LBMMessage.dataBuffer()**
- **com.latencybusters.lbm.LBMMessage.dataLength()**
- **com.latencybusters.lbm.LBMMessage.dispose()**
- **com.latencybusters.lbm.LBMObjectRecycler**
- **com.latencybusters.lbm.LBMObjectRecycler.doneWithMessage()**

These functions prevent unnecessary garbage and are strongly recommended by Informatica (see Zero Object Delivery).

The mySaveMessageForMoreProcessing() function is just whatever you use to transfer the message to your processing thread. The myAdditionalProcessing() function is typically called by you when you are ready to complete processing of the message.

The rules for the message processing code are the same as the event queue case. However, the application must ensure that every message promoted is eventually disposed.

If this is a Persistent receiver, see **Persistence Message Consumption**.

9.2.3 .NET Message Reception

Except where indicated, .NET coding for message reception is identical to that of Java.

Note

> Historically, .NET programs that receive UM messages were not required to call the message object's `dispose()` method. However, it is now strongly recommended. Not calling dispose() will introduce significant latency outliers (jitter) when GC runs, and also makes persistence acknowledgements to the Store non-deterministic. Finally, in the future, performance improvements for .NET might require the use of "dispose()". **Informatica strongly recommends that .NET subscribers call "dispose()" for every message.**

.NET: Message Is Fully Processed by the Application's Receiver Callback, Called by a Context or XSP Thread

```
public int onReceive(Object cbArg, LBMMessage msg)
{
  try {
    switch (msg.type())
    {
      case LBM.MSG_DATA:
        byte * message = msg.dataPointer();  // Different from java!
        uint msgLen = msg.length();          // Different from java!

        // Process "msgLen" bytes in "message".
        ...

        // Tell UM: finished with "msg".
        msg.dispose();
        break;

      /* Handle other receiver events. */
    }  /* switch */
  } catch (Exception e) {
    /* Handle exception. */
  }
  return 0;
}  /* onReceive */
```

This demonstrates the use of:

- `dataPointer()`
- `length()`
- `dispose()`

These functions prevent unnecessary garbage and are strongly recommended by Informatica (see Zero Object Delivery).

Important rules regarding the receiver callback function when called by the context thread:

- The function must return 0.
- The function must not be allowed to pass an unhandled exception back into UM. For example, you could have your entire callback function enclosed in a large try/catch. (This is true of all UM callbacks, not just receiver.)
- The passed-in message must not be modified.
- The passed-in message must be disposed. In .NET, every message should explicitly be disposed, to properly clean up the native memory. Do not assume that GC will clean it up.
- The function should not perform any operation that might be time consuming or put the thread to sleep (block). This is because any delays in your callback prevents the context thread from servicing its sockets, increasing the risk of packet loss.
- If the receiver callback sends a message, non-blocking sends must be used (see `SRC_NONBLOCK()`). And the code should be written to handle a send failure of `LBMEWouldBlockException()`.
- It is not allowed to create/delete sources or receivers, or subscribe/unsubscribe from a Spectrum channel from a callback function executed by the context thread.
- You may schedule UM timers.

If this is a Persistent receiver, see **Persistence Message Consumption**.

.NET: Message Is Fully Processed by the Application's Receiver Callback, Called by an Event Queue Dispatch Thread

The code is identical to the context thread case above.

The rules are similar, with some important differences (in bold):

- The function must return 0.
- The function must not be allowed to pass an unhandled exception back into UM. For example, you could have your entire callback function enclosed in a large try/catch. (This is true of all UM callbacks, not just receiver.)
- The passed-in message must not be modified.
- The passed-in message must be disposed. In .NET, every message should explicitly be disposed, to properly clean up the native memory. Do not assume that GC will clean it up.
- **The application is permitted to perform operations that are time consuming and/or blocking, as long as the average message processing rate is greater than the average message sending rate. Any temporary delays will buffer messages in the event queue.**
- **Blocking or non-blocking sends may be used, according to the application's preferences.**
- **You may create/delete sources or receivers, or subscribe/unsubscribe from a Spectrum channel. However, the receiver which is delivering the current message should not be deleted.**
- You may schedule UM timers.

- Informatica strongly recommends monitoring event queue length. See Event Queue Monitor for more information.

If this is a Persistent receiver, see **Persistence Message Consumption**.

.NET: Message Is Retained by the Receiver Callback for Further Processing by a Different Application Thread

```
public void init_UM()
{
  // During initialization, the UM context and receiver(s) must be
  // configured to use a recycler.
  LBMObjectRecycler objRec = new LBMObjectRecycler();
...
  // Preparing to create context - set up attribute.
  LBMContextAttributes ctx_attr = new LBMContextAttributes();
  ctx_attr.setObjectRecycler(objRec, null);
  // Proceed with creating context, using ctx_attr.
...
  // Preparing to create receiver - set up attribute.
  LBMReceiverAttributes rcv_attr = new LBMReceiverAttributes();
  rcv_attr.setObjectRecycler(objRec, null);
  // Proceed with creating receiver, using rcv_attr.
...
}

...

public int onReceive(Object cbArg, LBMMessage msg)
{
  try {
    switch (msg.type())
    {
      case LBM.MSG_DATA:
        msg.promote();
        // Tell UM: not finished with "msg".
        saveMessageForMoreProcessing(msg);
        break;

      /* Handle other receiver events. */
    }  /* switch */
  } catch (Exception e) {
    /* Handle exception. */
  }
  return 0;
}  /* onReceive */

...

public int myAdditonalProcessing(LBMMessage msg)
{
  byte * message = msg.dataPointer();  // Different from java!
  uint msgLen = msg.length();          // Different from java!

  // Process "msgLen" bytes in "message".
  ...

  // Tell UM: finished with "msg".
  msg.dispose();
  objRec.doneWithMessage(msg);
}  /* myAdditonalProcessing */
```

This demonstrates the use of:

- `promote()`
  - informs UM that the message is retained after the return of the receiver callback.
- `dataPointer()`
- `length()`
- `dispose()`
- `LBMObjectRecycler`
- `LBMObjectRecycler.doneWithMessage()`

These functions prevent unnecessary garbage and are strongly recommended by Informatica (see Zero Object Delivery).

The mySaveMessageForMoreProcessing() function is just whatever you use to transfer the message to your processing thread. The myAdditionalProcessing() function is typically called by you when you are ready to complete processing of the message.

The rules for the message processing code are the same as the event queue case. However, the application must ensure that every message promoted is eventually disposed.

If this is a Persistent receiver, see **Persistence Message Consumption**.

Chapter 10

UM Features

Except where otherwise indicated, the features described in this section are available in the UMS, UMP, and UMQ products.

10.1 Transport Services Provider (XSP)

As of UM version 6.11, a new receive-side object is available to the user: the Transport Services Provider Object.

The earlier feature, Multi-Transport Threads (MTT), is removed from UM in favor of XSP.

By default, a UM context combines all network data reception into a single *context thread*. This thread is responsible for reception and processing of application messages, topic resolution, and immediate message traffic (UIM and MIM). The context thread is also used for processing timers. This single-threaded model conserves CPU core resources, and can simplify application design. However, it can also introduce significant latency outliers (jitter) if a time-sensitive user message is waiting behind, say, a topic resolution message, or a timer callback.

Using an XSP object, an application can reassign the processing of a subscribed Transport Session to an independent thread. This allows concurrent processing of received messages with topic resolution and timers, and even allows different groups Transport Sessions to be processed concurrently with each other.

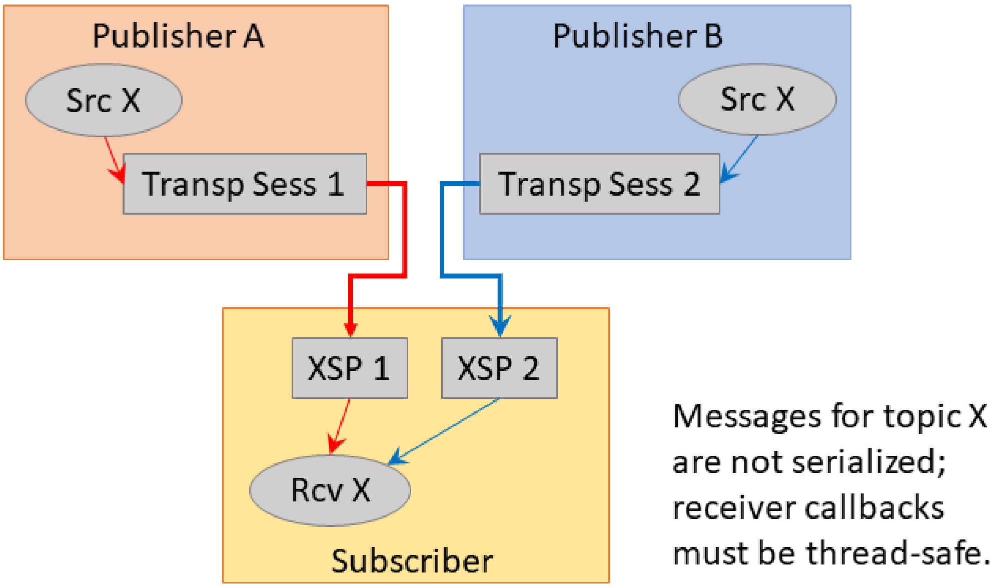
By default, when an XSP object is created, UM creates a new thread associated with the XSP. Alternatively, the XSP can be created with Sequential Mode, which gives the responsibility of thread creation to the application. Either way, the XSP uses its independent thread to read data from the sockets associated with one or more subscribed Transport Sessions. That thread then delivers received messages to the application via a normal receive application callback function.

Creation of an XSP does not by itself cause any receiver Transport Sessions to be assigned to it. Central to the use of XSPs is an application-supplied mapping callback function which tells UM which XSP to associate with subscribed Transport Sessions as they are discovered and joined. This callback allows the application to examine the newly-joined Transport Session, if desired. Then the callback returns, informing UM which XSP, if any, to assign the receiver Transport Session to.

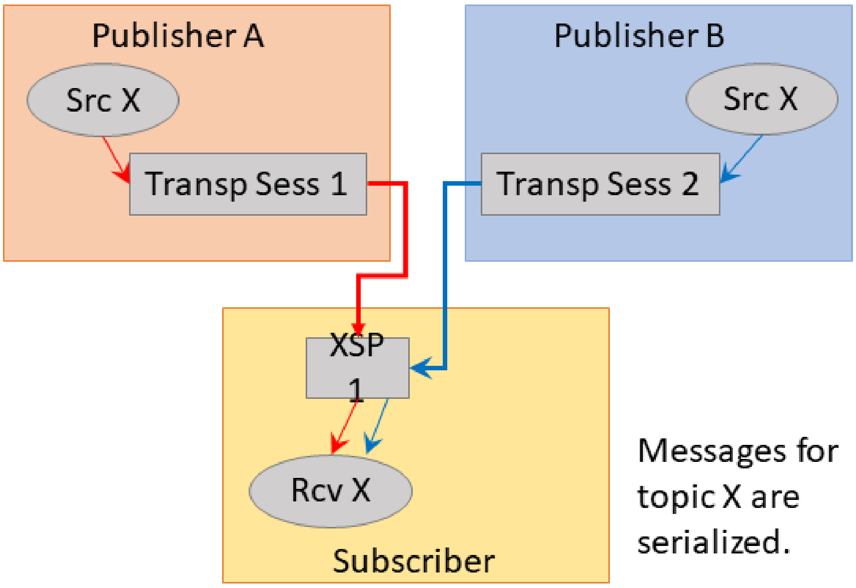
10.1.1 XSP Handles Transport Sessions, Not Topics

Conceptually, an application designer might want to assign the reception and processing of received data to XSPs on a topic basis. This is not always possible. The XSP thread must process received data on a socket basis, and sockets map to *Transport Sessions*. As mentioned in UM Transports, a publishing application maps one or more topic-based sources to a Transport Session.

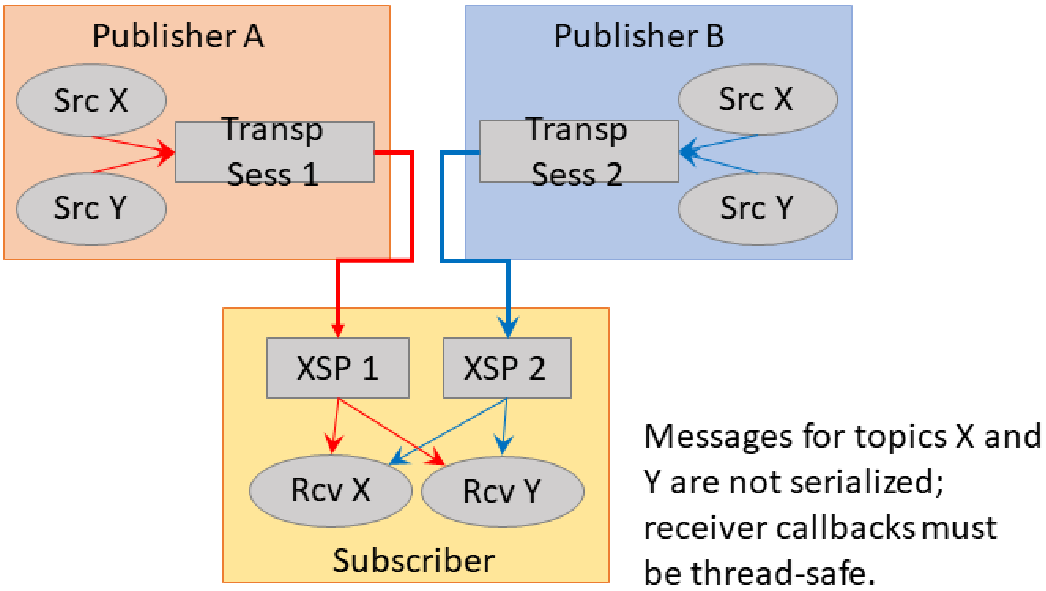
Consider the following example:



Publisher A and B are two separate application instances, both of which create a source for topic "X". A subscriber application might create two XSPs and assign one Transport Session to each. In this case, you have two independent threads delivering messages to the subscriber's receiver callback, which may not be what the developer wanted. If the developer wants topic X to be serialized, a single XSP should be created and mapped to both Transport Sessions:



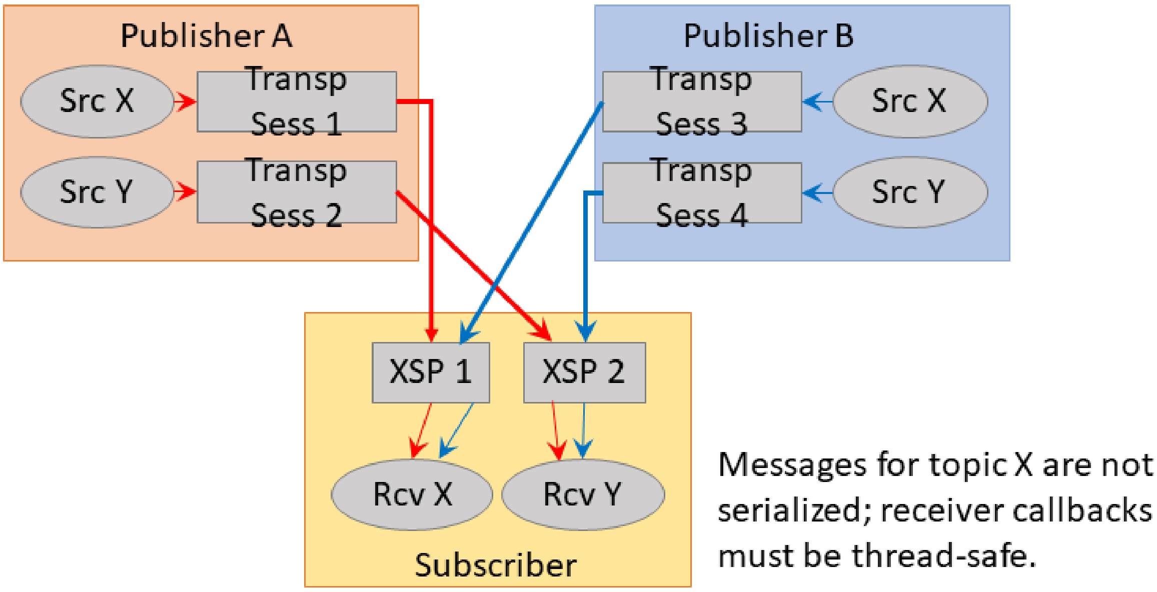
Now let's introduce a second topic. The developer might want to create two XSPs so that each topic will be handled by an independent thread. However, this is not possible, given the way that the topics are mapped to Transport Sessions in the following example:



Messages for topics X and Y are not serialized; receiver callbacks must be thread-safe.

In this case, XSP 1 is delivering both topics X and Y from Publisher A, and XSP 2 is delivering topics X and Y from Publisher B. Once again, the receiver callback for topic X will be called by two independent threads, which is not desired.

The only way to achieve independent processing of topics is to design the publishers to map their topics to Transport Sessions carefully. For example:



Messages for topic X are not serialized; receiver callbacks must be thread-safe.

10.1.2 XSP Threading Considerations

When contexts are used single-threaded, the application programmer can assume serialization of event delivery to the application callbacks. This can greatly simplify the design of applications, at the cost of added latency outliers (jitter).

When XSPs are used to provide multi-threaded receivers, care must be taken in application design to account for potential concurrent calls to application callbacks. This is especially true if multiple subscribed Transport Sessions are assigned different XSPs, as demonstrated in XSP Handles Transport Sessions, Not Topics.

Even in the most simple case, where a single XSP is created and used for all subscribed Transport Sessions, there are still events generated by the main context thread which can be called concurrently with XSP callbacks. Reception of MIM or UIM messages, scheduled timers, and some topic resolution-related callbacks all come from the main context thread, and can all be invoked concurrently with XSP callbacks.

Warning

> Remember that MIM and UIM messages can be sent to a topic. If you have regular sources for a topic assigned to an XSP, and somebody sends MIM or UIM messages to the same topic, your receiver callback function can be called concurrently by both the XSP and the main context. Informatica recommends having a given topic sent to by only one type of sender (UIM, MIM, Source).

Threading Example: Message Timeout

Consider as an example a common timer use case: message timeout. Application A expects to receive messages for topic "X" every 5 seconds. If 10 seconds pass without a message, the application assumes that the publisher for "X" has exited, so it cleans up internal state and deletes the UM receiver object. Each time a message is received, the current timer is cancelled and re-created for 10 seconds.

Without XSPs, this can be easily coded since message reception and timer expiration events are serialized. The timer callback can clean up and delete the receiver, confident that no receiver events might get delivered while this is in progress.

However, if the Transport Session carrying topic "X" is assigned to an independent XSP thread, message reception and timer expiration events are no longer serialized. Publisher of "X" might send it's message on-time, but a temporary network outage could delay its delivery, introducing a race condition between message delivery and timer expiration. Consider the case where the timer expiration is a little ahead of the message callback. The timer callback might clean up application state which the message callback will attempt to use. This could lead to unexpected behavior, possibly including segmentation faults.

In this case, proper sequencing of operations is critical. The timer should delete the receiver first. While inside the receiver delete API, the XSP might deliver messages to the application. However, once the receiver delete API returns, it is guaranteed that the XSP is finished making receiver callbacks.

Note that in this example case, if the message receiver callback attempts to cancel the timer, the cancel API will return an error. This is because the timer has already expired and the execution of the callback has begun, and is inside the receiver delete API. The message receiver callback needs to be able to handle this sequence, presumably by not re-scheduling the timer.

10.1.3 XSP Usage

This section provides simplified C code fragments that demonstrate some of the XSP-related API calls. For full examples of XSP usage, see **Example lbmrcvxsp.c** (for C) and **Example lbmrcvxsp.java** (for Java).

Note

> Each XSP thread has its own Unicast Listener (request) port. You may need to expand the range **request_tcp_port_low (context)** - **request_tcp_port_high (context)**.

The common sequence of operations during application initialization is minimally shown below. In the code fragments below, error detection and handling are omitted for clarity.

1. Create a context attribute object and set the **transport_mapping_function (context)** option to point at the application's XSP mapping callback function using the structure **lbm_transport_mapping_func_t**.

```
lbm_context_attr_t *ctx_attr;
err = lbm_context_attr_create_from_xml(&ctx_attr, "MyCtx");

lbm_transport_mapping_func_t mapping_func;
mapping_func.mapping_func = app_xsp_mapper_callback;
mapping_func.clientd = NULL;             /* Can include app state pointer. */

err = lbm_context_attr_setopt(ctx_attr, "transport_mapping_function",
        &mapping_func, sizeof(mapping_func));
```

2. Create the context.

   ```
   err = lbm_context_create(&ctx, ctx_attr, NULL, NULL);
   err = lbm_context_attr_delete(ctx_attr);  /* No longer needed. */
   ```

3. Create XSPs using **lbm_xsp_create()**. In this example, only a single XSP is created.

   ```
   lbm_xsp_t *xsp;  /* app_xsp_mapper_callback() needs this; see below. */
   err = lbm_xsp_create(&xsp, ctx, NULL, NULL);
   ```

   Note that the application can optionally pass in a context attribute object and an XSP attribute object. The context attribute is because XSP is implemented as a sort of reduced-function sub-context, and so it is possible to modify context options for the XSP. However, this is rarely needed since the default action is for the XSP to inherit all the configuration of the main context.

4. Create a receiver for topic "X".

   ```
   lbm_topic_t *topic;
   err = lbm_rcv_topic_lookup(&topic, ctx, "X", NULL);

   lbm_rcv_t *rcv;
   err = lbm_rcv_create(&rcv, ctx, topic, app_rcv_callback, NULL, NULL);
   ```

   Event queues may also be used with XSP-assigned Transport Sessions.

5. At this point, when the main context discovers a source for topic "X", it will proceed to join the Transport Session. It will call the application's app_xsp_mapper_callback() function, which is minimally this:

   ```
   lbm_xsp_t *app_xsp_mapper_callback(lbm_context_t *ctx,
           lbm_new_transport_info_t *transp_info, void *clientd)
   {
     /* Retrieve the XSP object created in step 3. */
     return xsp;
   }
   ```

   This minimal callback simply returns the XSP that was created during initialization (the "clientd" can be helpful for that). By assigning all receiver Transport Sessions to the same XSP, you have effectively separated message processing from UM housekeeping tasks, like processing of topic resolution and timers. This can greatly reduce latency outliers (jitter).

   As described in XSP Handles Transport Sessions, Not Topics, some users want to have multiple XSPs and assign the Transport Sessions to XSPs according to application logic. Note that the passed-in **lbm_new_transport_info_t** structure contains information about the Transport Session, such as the IP address of the sender. However, this structure does not contain topic information. Applications can use the resolver's source notification callback via the **resolver_source_notification_function (context)** option to associate topics with source strings.

Note

> Most of the time, the application mapping callback will be invoked each time a Transport Session is joined. However, there is one exception to this rule. If a context is already joined to a Transport Session carried on a multicast group and destination port, joining another Transport Session on the same multicast group and destination port does not invoke the mapping callback again. This is because the same socket is used for all Transport Sessions that use the same group:port.

10.1.4 Other XSP Operations

As of UM 6.12, XSP supports persistent receivers.

When an XSP object is created, an XSP attribute object can be supplied to set XSP options. The XSP options are:

- **operational_mode (xsp)**
- **zero_transports_function (xsp)**

To create and manipulate an XSP attribute object, see:

- **lbm_xsp_attr_setopt()**
- **lbm_xsp_attr_getopt()**
- **lbm_xsp_attr_delete()**

To delete an XSP, all receivers associated with Transport Sessions handled by that XSP must first be deleted. Then the XSP can be deleted using **lbm_xsp_delete()**.

XSP supports Registered File Descriptors.

10.1.5 XSP Limitations

There are some restrictions and limitations on the XSP feature.

- The only transport types currently supported are LBT-RM, LBT-RU, and TCP. The XSP feature is not compatible with transport types IPC, SMX, or DBL.
- An application receiver callback must not create a new XSP.
- For a persistent receiver assigned to an XSP, **ume_proactive_keepalive_interval (context)** must be enabled.
- XSP is not currently compatible with **Ultra Load Balancing (ULB)**.
- XSP is not currently compatible with Hot Failover (HF).
- XSP is not currently compatible with Receive-Side Batching.

10.2 Using Late Join

This section introduces the use of Ultra Messaging Late Join in default and specialized configurations. See **Late Join Options** for more information.

Note

> If your application is running within a Ultra Messaging context with configuration option **request_tcp_bind_request_port (context)** set to zero, then UIM port binding has been turned off, which also disables the Late Join feature.
> With the UMQ product, you cannot use Late Join with Queuing (both Brokered and ULB).

The Late Join feature enables newly created receivers to receive previously transmitted messages. Sources configured for Late Join maintain a retention buffer (not to be confused with a transport retransmission window), which holds transmitted messages for late-joining receivers.

A Late Join operation follows the following sequence:

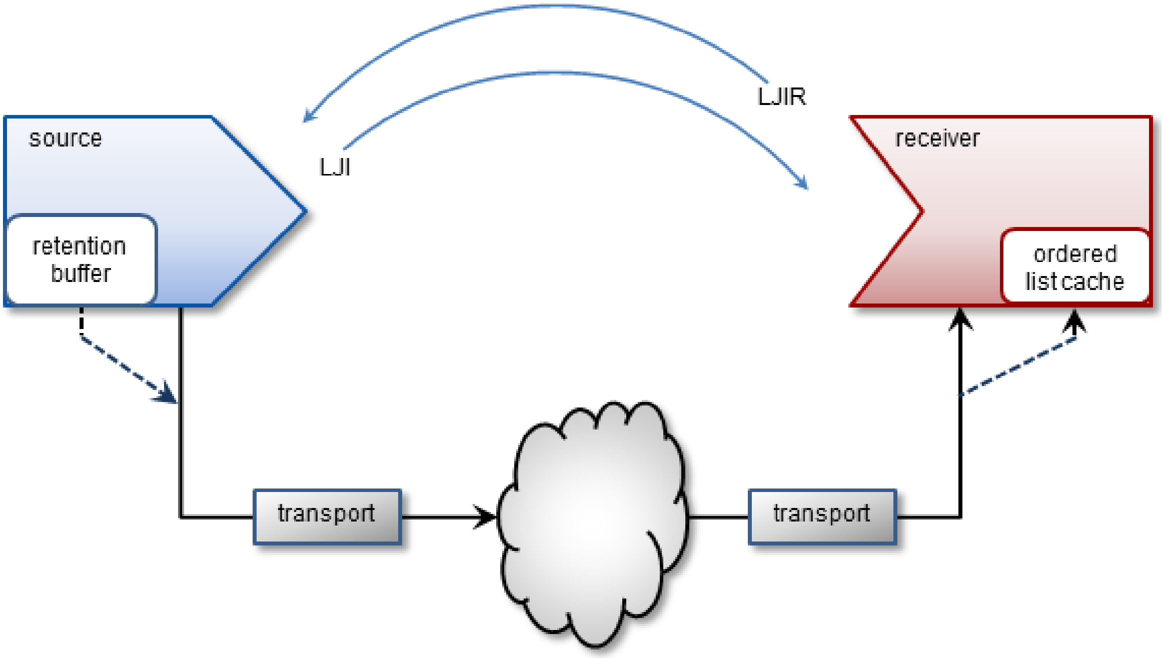
1. A new receiver configured for Late Join with **use_late_join (receiver)** completes topic resolution. Topic advertisements from the source contain a flag that indicates the source is configured for Late Join with **late↩ _join (source)**.

2. The new receiver sends a Late Join Information Request (LJIR) to request a previously transmitted messages. The receiver configuration option, **retransmit_request_outstanding_maximum (receiver)**, determines the number of messages the receiver requests.

3. The source responds with a Late Join Information (LJI) message containing the sequence numbers for the retained messages that are available for retransmission.

4. The source unicasts the messages.

5. When Configuring Late Join for Large Numbers of Messages, the receiver issues additional requests, and the source retransmits these additional groups of older messages, oldest first.



The source's retention buffer's is not pre-allocated and occupies an increasing amount of memory as the source sends messages and adds them to the buffer. If a retention buffer grows to a size equal to the value of the source configuration option, **retransmit_retention_size_threshold (source)**, the source deletes older messages as it adds new ones. The source configuration option **retransmit_retention_age_threshold (source)**, controls message deletion based on message age.

UM uses control-structure overhead memory on a per-message basis for messages held in the retention buffer, in addition to the retention buffer's memory. Such memory usage can become significantly higher when retained messages are smaller in size, since more of them can then fit in the retention buffer.

Note

> If you set the receiver configuration option **ordered_delivery (receiver)** to 1, the receiver must deliver messages to your application in sequence number order. The receiver holds out-of-order messages in an ordered list cache until messages arrive to fill the sequence number gaps. If an out-of-order message arrives with a sequence number that creates a message gap greater than the value of **retransmit_message_caching_↩ proximity (receiver)**, the receiver creates a burst loss event and terminates the Late Join recovery operation. You can increase the value of the proximity option and restart the receiver, but a burst loss is a significant event and you should investigate your network and message system components for failures.

10.2.1 Late Join With Persistence

With the UMP/UMQ products, late Join can be implemented in conjunction with the persistent Store, however in this configuration, it functions somewhat differently from Streaming. After a late-Join-enabled receiver has been created, resolved a topic, and become registered with a Store, it may then request older messages. The Store unicasts the retransmission messages. If the Store does not have these messages, it requests them of the source (assuming option retransmission-request-forwarding is enabled), thus initiating Late Join.

10.2.2 Late Join Options Summary

- **late_join (source)**
- **retransmit_retention_age_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_size_threshold (source)**
- **use_late_join (receiver)**
- **retransmit_initial_sequence_number_request (receiver)**
- **retransmit_message_caching_proximity (receiver)**
- **retransmit_request_message_timeout (receiver)**
- **retransmit_request_interval (receiver)**
- **retransmit_request_maximum (receiver)**
- **retransmit_request_outstanding_maximum (receiver)**

10.2.3 Using Default Late Join Options

To implement Late Join with default options, set the Late Join configuration options to activate the feature on both a source and receiver in the following manner.

1. Create a configuration file with source and receiver Late Join activation options set to 1. For example, file cfg1.cfg containing the two lines:

```
source late_join 1
receiver use_late_join 1
```

2. Run an application that starts a Late-Join-enabled source. For example:

   ```
   lbmsrc -c cfg1.cfg -P 1000 topicName
   ```

3. Wait a few seconds, then run an application that starts a Late-Join-enabled receiver. For example:

   ```
   lbmrcv -c cfg1.cfg -v topicName
   ```

The output for each should closely resemble the following:

LBMSRC

```
$ lbmsrc -c cfg1.cfg -P 1000 topicName
LOG Level 5: NOTICE: Source "topicName" has no retention settings (1 message
    retained max)
Sending 10000000 messages of size 25 bytes to topic [topicName]
Receiver connect [TCP:10.29.3.77:34200]
```

LBMRCV

```
$ lbmrcv -c cfg1.cfg -v topicName
Immediate messaging target: TCP:10.29.3.77:4391
[topicName][TCP:10.29.3.76:4371][2]-RX-, 25 bytes
1.001 secs. 0.0009988 Kmsgs/sec. 0.1998 Kbps
[topicName][TCP:10.29.3.76:4371][3], 25 bytes
1.002 secs. 0.0009982 Kmsgs/sec. 0.1996 Kbps
[topicName][TCP:10.29.3.76:4371][4], 25 bytes
1.003 secs. 0.0009972 Kmsgs/sec. 0.1994 Kbps
[topicName][TCP:10.29.3.76:4371][5], 25 bytes
1.003 secs. 0.0009972 Kmsgs/sec. 0.1994 Kbps
...
```

Note that the source only retained 1 Late Join message (due to default retention settings) and that this message appears as a retransmit (-RX-). Also note that it is possible to sometimes receive 2 RX messages in this scenario (see Retransmitting Only Recent Messages.)

10.2.4 Specifying a Range of Messages to Retransmit

To receive more than one or two Late Join messages, increase the source's **retransmit_retention_size_threshold (source)** from its default value of 0. Once the buffer exceeds this threshold, the source allows the next new message entering the retention buffer to bump out the oldest one. Note that this threshold's units are bytes (which includes a small overhead per message).

While the retention threshold endeavors to keep the buffer size close to its value, it does not set hard upper limit for retention buffer size. For this, the **retransmit_retention_size_limit (source)** configuration option (also in bytes) sets this boundary.

Follow the steps below to demonstrate how a source can retain about 50MB of messages, but no more than 60MB:

1. Create a second configuration file (cfg2.cfg) with the following options:

   ```
   source late_join 1
   source retransmit_retention_size_threshold 50000000
   source retransmit_retention_size_limit 60000000
   receiver use_late_join 1
   ```

2. Run lbmsrc -c cfg2.cfg -P 1000 topicName.

3. Wait a few seconds and run `lbmrcv -c cfg2.cfg -v topicName`. The output for each should closely resemble the following:

LBMSRC

```
$ lbmsrc -c cfg2.cfg -P 1000 topicName
Sending 10000000 messages of size 25 bytes to topic [topicName]
Receiver connect [TCP:10.29.3.76:34444]
```

LBMRCV

```
$ lbmrcv -c cfg2.cfg -v topicName
Immediate messaging target: TCP:10.29.3.76:4391
[topicName][TCP:10.29.3.77:4371][0]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][1]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][2]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][3]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][4]-RX-, 25 bytes
1.002 secs. 0.004991 Kmsgs/sec. 0.9981 Kbps
[topicName][TCP:10.29.3.77:4371][5], 25 bytes
1.002 secs. 0.0009984 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][6], 25 bytes
1.002 secs. 0.0009983 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][7], 25 bytes
...
```

Note that lbmrcv received live messages with sequence numbers 7, 6, and 5, and RX messages going from 4 all the way back to Sequence Number 0.

10.2.5 Retransmitting Only Recent Messages

Thus far we have worked with only source late join settings, but suppose that you want to receive only the last 10 messages. To do this, configure the receiver option **retransmit_request_maximum (receiver)** to set how many messages to request backwards from the latest message.

Follow the steps below to set this option to 10.

1. Add the following line to cfg2.cfg and rename it cfg3.cfg:

   ```
   receiver retransmitrequestmaximumreceiver 10
   ```

2. Run:

   ```
   lbmsrc -c cfg3.cfg -P 1000 topicName
   ```

3. Wait a few seconds and run `lbmrcv -c cfg3.cfg -v topicName`. The output for each should closely resemble the following.

LBMSRC

```
$ lbmsrc -c cfg3.cfg -P 1000 topicName
Sending 10000000 messages of size 25 bytes to topic [topicName]
Receiver connect [TCP:10.29.3.76:34448]
```

LBMRCV

```
$ lbmrcv -c cfg3.cfg -v topicName
Immediate messaging target: TCP:10.29.3.76:4391
[topicName][TCP:10.29.3.77:4371][13]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][14]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][15]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][16]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][17]-RX-, 25 bytes
```

```
[topicName][TCP:10.29.3.77:4371][18]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][19]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][20]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][21]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][22]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][23]-RX-, 25 bytes
1.002 secs. 0.01097 Kmsgs/sec. 2.195 Kbps
[topicName][TCP:10.29.3.77:4371][24], 25 bytes
1.002 secs. 0.0009984 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][25], 25 bytes
1.002 secs. 0.0009984 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][26], 25 bytes
...
```

Note that 11, not 10, retransmits were actually received. This can happen because network and timing circumstances may have one RX already in transit while the specific RX amount is being processed. (Hence, it is not possible to guarantee one and only one RX message for every possible Late Join recovery.)

10.2.6 Configuring Late Join for Large Numbers of Messages

Suppose you have a persistent receiver that comes up at midday and must gracefully catch up on the large number of messages it has missed. The following discussion explains the relevant Late Join options and how to use them. (The discussion also applies to streaming-based late join, but since streaming sources must hold all retained messages in memory, there are typically far fewer messages available.)

Option: retransmit_request_outstanding_maximum (receiver)

When a receiver comes up and begins requesting Late Join messages, it does not simply request messages starting at Sequence Number 0 through 1000000. Rather, it requests the messages a little at a time, depending upon how option **retransmit_request_outstanding_maximum (receiver)** is set. For example, when set to the default of 10, the receiver sends requests the first 10 messages (Sequence Number 0 - 9). Upon receiving Sequence Number 0, it then requests the next message (10), and so on, limiting the number of outstanding unfulfilled requests to 10.

Note that higher for values retransmit_request_outstanding_maximum can increase the rate of RXs received, which can reduce the time required for receiver recovery. However, this can lead to heavy loading of the Store, potentially making it unable to sustain the incoming data rate.

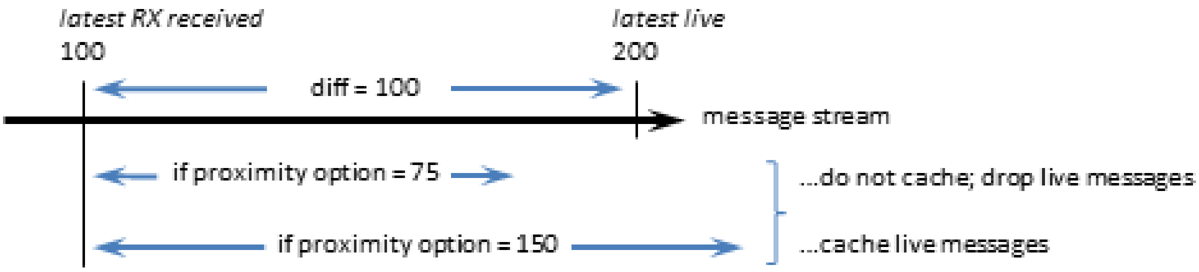
Also, be aware that increasing retransmit_request_outstanding_maximum may require a corresponding increase to **retransmit_request_interval (receiver)**. Otherwise you can have a situation where messages time out because it takes a Store longer than retransmit_request_interval to process all retransmit_request_outstanding_maximum requests. When this happens, you can see messages needlessly requested and sent many times (generates warnings to the receiver application log file).

Option: retransmit_message_caching_proximity (receiver)

When sequence number delivery order is used, long recoveries of active sources can create receiver memory cache problems due to the processing of both new and retransmitted messages. This option provides a method to control caching and cache size during recovery.

It does this by comparing the option value (default 2147483647) to the difference between the newest (live) received sequence number and the latest received RX sequence number. If the difference is less than the option's value, the receiver caches incoming live new messages. Otherwise, new messages are dropped and not cached (with the assumption that they can be requested later as retransmissions).

For example, as shown in the diagram below, a receiver may be receiving both live streaming messages (latest, #200) and catch-up retransmissions (latest, #100). The difference here is 100. If **retransmit_message_caching↩_proximity (receiver)** is 75, the receiver caches the live messages and will deliver them when it is all caught up with the retransmissions. However, if this option is 150, streamed messages are dropped and later picked up again as a retransmission.



The default value of this option is high enough to still encourage caching most of the time, and should be optimal for most receivers.

If your source streams faster than it retransmits, caching is beneficial, as it ensures new data are received only once, thus reducing recovery time. If the source retransmits faster than it streams, which is the optimal condition, you can lower the value of this option to use less memory during recovery, with little performance impact.

10.3 Off-Transport Recovery (OTR)

Off-Transport Recovery (OTR) is a lost-message-recovery feature that provides a level of hedging against the possibility of brief and incidental unrecoverable loss at the transport level or from a DRO. This section describes the OTR feature.

Note

> With the UMQ product, you cannot use OTR with Queuing (both Brokered and ULB).

When a transport cannot recover lost messages, OTR engages and looks to the source for message recovery. It does this by accessing the source's retention buffer (used also by the Late Join feature) to re-request messages that no longer exist in a transport's transmission window, or other places such as a persistent Store or redundant source.

OTR functions in a manner very similar to that of Late Join, but differs mainly in that it activates in message loss situations rather than following the creation of a receiver, and shares only the **late_join (source)** option setting.

Upon detecting loss, a receiver initiates OTR by sending repeated, spaced, OTR requests to the source, until it recovers lost messages or a timeout period elapses.

OTR operates independently from transport-level recovery mechanisms such as NAKs for LBT-RU or LBT-RM. When you enable OTR for a receiver with **use_otr (receiver)**, the **otr_request_initial_delay (receiver)** period starts as soon as the Delivery Controller detects a sequence gap. If the gap is not resolved by the end of the delay interval, OTR recovery initiates. OTR recovery can occur before, during or after transport-level recovery attempts.

When a receiver initiates OTR, the intervals between OTR requests increases twofold after each request, until the maximum interval is reached (assuming the receiver is still waiting to receive the retransmission). You use configuration options **otr_request_minimum_interval (receiver)** and **otr_request_maximum_interval (receiver)** to set the initial (minimum) and maximum intervals, respectively.

The source retransmits lost messages to the recovered receiver via unicast.

10.3.1 OTR with Sequence Number Ordered Delivery

When sequence number delivery order is used and a gap of missing messages occurs, a receiver buffers the new incoming messages while it attempts to recover the earlier missing ones. Long recoveries of actively streaming sources can cause excessive receiver cache memory growth due to the processing of both new and retransmitted messages. You can control caching and cache size during recovery with options **otr_message_caching_threshold (receiver)** and **retransmit_message_caching_proximity (receiver)**.

The option **otr_message_caching_threshold (receiver)** sets the maximum number of messages a receiver can buffer. When the number of cached messages hits this threshold, new streamed messages are dropped and not cached, with the assumption that they can be requested later as retransmissions.

The **retransmit_message_caching_proximity (receiver)**, which is also used by Late Join (see **retransmit_↩ message_caching_proximity (receiver)**), turns off this caching if there are too many messages to buffer between the last delivered message and the currently streaming messages.

Both of these option thresholds must be satisfied before caching resumes.

10.3.2 OTR With Persistence

With the UMP/UMQ products, you can implement OTR in conjunction with the persistent Store, however in this configuration, it functions somewhat differently from Streaming. If an OTR-enabled receiver registered with a Store detects a sequence gap in the live stream and that gap is not resolved by other means within the next **otr_request↩ _initial_delay (receiver)** period, the receiver requests those messages from the Store(s). If the Store does not have some of the requested messages, the receiver requests them from the source. Regardless of whether the messages are recovered from a Store or from the source, OTR delivers all recovered messages with the LBM_MSG_OTR flag, unlike Late Join, which uses the LBM_MSG_RETRANSMIT flag.

10.3.3 OTR Options Summary

- **late_join (source)**
- **retransmit_retention_age_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_size_threshold (source)**
- **use_otr (receiver)**
- **otr_request_message_timeout (receiver)**
- **otr_request_initial_delay (receiver)**
- **otr_request_log_alert_cooldown (receiver)**
- **otr_request_maximum_interval (receiver)**
- **otr_request_minimum_interval (receiver)**
- **otr_request_outstanding_maximum (receiver)**
- **otr_message_caching_threshold (receiver)**
- **retransmit_message_caching_proximity (receiver)**

Note

With Smart Sources, the following configuration options have limited or no support:

- **retransmit_retention_size_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_age_threshold (source)**

10.4 Encrypted TCP

This section introduces the use of Transport Layer Security (TLS), sometimes known by its older designation Secure Sockets Layer (SSL).

The goal of the Ultra Messaging (UM) TLS feature is to provide encrypted transport of application data. TLS supports authentication (through certificates), data confidentiality (through encryption), and data integrity (ensuring data are not changed, removed, or added-to). UM can be configured to apply TLS security measures to all Streaming and/or Persisted TCP communication, including DRO peer links. Non-TCP communication is not encrypted (e.g. topic resolution).

TLS is a family of standard protocols and algorithms for securing TCP communication between a client and a server. It is sometimes referred as "SSL", which technically is the name of an older (less secure) version of the protocol. Over the years, security researchers (and hackers) have discovered flaws in SSL/TLS. However, the vast majority of the widely publicized security vulnerabilities have been flaws in the implementations of TLS, not in the recent TLS protocols or algorithms themselves. As of UM version 6.9, there are no known security weaknesses in TLS version 1.2, the version used by UM.

TLS is generally implemented by several different software packages. UM makes use of OpenSSL, a widely deployed and actively maintained open-source project.

10.4.1 TLS Authentication

TLS authentication uses X.509 digital certificates. Certificate creation and management is the responsibility of the user. Ultra Messaging's usage of OpenSSL expects PEM encoded certificates. There are a variety of generally available tools for converting certificates between different encodings. Since user infrastructures vary widely, the UM package does not include tools for creation, formatting, or management of certificates.

Although UM is designed as a peer-to-peer messaging system, TLS has the concept of client and server. The client initiates the TCP connection and the server accepts it. In the case of a TCP source, the receiver initiates and is therefore the client, with the source (sender of data) being the server. However, with unicast immediate messages, the sender of data is the client, and the recipient is the server. Due to the fact that unicast immediate messages are used by UM for internal control and coordination, it is typically not possible to constrain a given application to only operate as a pure client or pure server. For this reason, UM requires all applications participating in encryption to have a certificate. Server-only authentication (i.e. anonymous client, as is used by web browsers) is not supported. It is permissible for groups of processes, or even all processes, to share the same certificate.

A detailed discussion of certificate usage is beyond the scope of the Ultra Messaging documentation. However, you can find a step-by-step procedure for creating a self-signed X.509 security certificate here: https://kb.informatica.com/howto/6/Pages/18/432752.aspx

10.4.2 TLS Backwards Compatibility

The TLS protocol was designed to allow for a high degree of backwards compatibility. During the connection establishment phase, the client and server perform a negotiation handshake in which they identify the highest common versions of various security options. For example, an old web browser might pre-date the introduction of TLS and only support the older SSL protocol. OpenSSL is often configured to allow clients and servers to "negotiate down" to those older, less-secure protocols or algorithms.

Ultra Messaging has the advantage of not needing to communicate with old versions of SSL or TLS. UM's default configuration directs OpenSSL to require both the client and the server to use protocols and algorithms which were highly regarded, as of UM's release date. If vulnerabilities are discovered in the future, the user can override UM's defaults and chose other protocols or algorithms.

10.4.3 TLS Efficiency

When a TLS connection is initiated, a handshake takes place prior to application data encryption. Once the handshake is completed, the CPU effort required to encrypt and decrypt application data is minimal. However, the handshake phase involves the use of much less efficient algorithms.

There are two factors under the user's control, which greatly affect the handshake efficiency: the choice of cipher suite and the key length. We have seen an RSA key of 8192 bits take 4 seconds of CPU time on a 1.3GHz SparcV9 processor just to complete the handshake for a single TLS connection.

Users should make their choices with an understanding of the threat profiles they are protecting against. For example, it is estimated that a 1024-bit RSA key can be broken in about a year by brute force using specialized hardware (see `http://www.tau.ac.il/~tromer/papers/cbtwirl.pdf`). This may be beyond the means of the average hacker, but well within the means of a large government. RSA keys of 2048 bits are generally considered secure for the foreseeable future.

10.4.4 TLS Configuration

TLS is enabled on a context basis. When enabled, all Streaming and Persistence related TCP-based communication into or out of the context is encrypted by TLS. A context with TLS enabled will not accept source creation with transports other than TCP.

Subscribers will only successfully receive data if the receiver's context and the source's context share the same encryption settings. A receiver created in an encrypted enabled context will ignore topic resolution source advertisements for non-encrypted sources, and will therefore not subscribe. Similarly, a receiver created in a non-encrypted context will ignore topic resolution source advertisements for encrypted sources. Topic resolution queries are also ignored by mismatched contexts. No warning will be logged when these topic resolution datagrams are ignored, but each time this happens, the context-level statistic tr_dgrams_dropped_type is incremented.

TLS is applied to unicast immediate messages as well, as invoked either directly by the user, or internally by functions like late join, request/response, and Persistence-related communication between sources, receivers, and Stores.

Brokered Queuing using AMQP does not use the UM TLS feature. A UM brokered context does not allow TLS to be enabled.

10.4.5 TLS Options Summary

- **use_tls (context)**
- **tls_cipher_suites (context)**
- **tls_certificate (context)**
- **tls_certificate_key (context)**
- **tls_certificate_key_password (context)**
- **tls_trusted_certificates (context)**
- **tls_compression_negotiation_timeout (context)**

The **tls_cipher_suites (context)** configuration option defines the list of one or more (comma separated) cipher suites that are acceptable to this context. If more than one is supplied, they should be in descending order of preference. When a remote context negotiates encrypted TCP, the two sides must find a cipher suite in common, otherwise the connection will be canceled.

OpenSSL uses the cipher suite to define the algorithms and key lengths for encrypting the data stream. The choice of cipher suite is critical for ensuring the security of the connection. To achieve a high degree of backwards compatibility, OpenSSL supports old cipher suites which are no longer considered secure. The user is advised to use UM's default suite.

OpenSSL follows its own naming convention for cipher suites. See OpenSSL's `Cipher Suite Names` for the full list of suite names. When configuring UM, use the OpenSSL names (with dashes), not* the IANA names (with underscores).

10.4.6 TLS and Persistence

TLS is designed to encrypt a TCP connection, and works with TCP-based persisted data Transport Sessions and control traffic. However, TLS is not intended to encrypt data at rest. When a persistent Store is used with the UM TLS feature, the user messages are written to disk in plaintext form, not encrypted.

10.4.7 TLS and Queuing

The UM TLS feature does not apply to the AMQP connection to the brokered queue. UM does not currently support security on the AMQP connection.

However, the ULB form of queuing does not use a broker. For ULB sources that are configured for TCP, the UM TLS feature will encrypt the application data.

10.4.8 TLS and the DRO

When a DRO is used to route messages across Topic Resolution Domains (TRDs), be aware that the TLS session is terminated at the DRO's proxy receiver/source. Because each endpoint portal on a DRO is implemented with its own context, care must be taken to ensure end-to-end security. It is possible to have a TLS source publishing in one TRD, received by a DRO (via an endpoint portal also configured for TLS), and re-published to a different TRD via an endpoint portal configured with a non-encrypted context. This would allow a non-encrypted receiver to access messages that the source intended to be encrypted. As a message is forwarded through a DRO network, it does not propagate the security settings of the originator, so each portal needs to be appropriately encrypted. The user is strongly encouraged to configure ALL portals on an interconnected network of DROs with the same encryption settings.

The encryption feature is extended to DRO peer links, however peer links are not context-based and are not configured the same way. The following XML elements are used by the DRO to configure a peer link:

- **Router Element "<tls>"**
- **Router Element "<cipher-suites>"**
- **Router Element "<certificate>"**
- **Router Element "<certificate-key>"**
- **Router Element "<certificate-key-password>"**
- **Router Element "<trusted-certificates>"**

As with sources and receivers, the portals on both sides of a peer link must be configured for compatible encryption settings.

Notice that there is no DRO element corresponding to the context option **tls_compression_negotiation_timeout (context)**. The DRO peer link's negotiation timeout is hard-coded to 5 seconds.

See **DRO Configuration DTD** for details.

10.4.9 TLS and Compression

Many users have advanced network equipment (switches/routers), which transparently compress packets as they traverse the network. This compression is especially valued to conserve bandwidth over long-haul WAN links. However, when packets are encrypted, the network compressors are typically not able to reduce the size of the data. If the user desires UM messages to be compressed and encrypted, the data needs to be compressed before it is encrypted.

The UM compression feature (see Compressed TCP) accomplishes this. When both TLS and compression are enabled, the compression is applied to user data first, then encryption.

Be aware that there can be information leakage when compression is applied and an attacker is able to inject data of known content over a compressed and encrypted session. For example, this leakage is exploited by the CRIME attack, albeit primarily for web browsers. Users must weigh the benefits of compression against the potential risk of information leakage.

Version Interoperability

It is not recommended to mix pre-6.9 contexts with encrypted contexts on topics of shared interest. If a process with a pre-6.9 version of UM creates a receiver, and another process with UM 6.9 or beyond creates a TLS source, the pre-6.9 receiver will attempt to join the TLS source. After a timeout, the handshake will fail and the source will disconnect. The pre-6.9 receiver will retry the connection, leading to flapping.

Note that in the reverse situation, a 6.9 TLS receiver will simply ignore a pre-6.9 source. I.e. no attempt will be made to join, and no flapping will occur.

10.4.10 OpenSSL Dependency

For UM versions 6.9 through 6.12, the UM dynamic library was linked with OpenSSL in such a way as to require its presence in order for UM to load and initialize. I.e. it was a load-time dependency.

As of UM version 6.12.1, the linkage with OpenSSL is made at run-time. If encryption features are not used, the OpenSSL libraries do not need to be present on the system. UM is able to initialize without OpenSSL.

There are two UM features which utilize encryption provided by OpenSSL:

- Encrypted TCP.
- **UM Manager (UMM)** with **secure communication** enabled.

10.5 Compressed TCP

This section introduces the use of Compression with TCP connections.

The goal of the Ultra Messaging (UM) compression feature is to decrease the size of transmitted application data. UM can be configured to apply compression to all Streaming and/or Persisted TCP communication.

Non-TCP communication is not compressed (e.g. topic resolution).

Compression is generally implemented by any of several different software packages. UM makes use of LZ4, a widely deployed open-source project.

While the UM compression feature is usable for TCP-based sources and receivers, it is possibly most useful when applied to DRO peer links.

10.5.1 Compression Configuration

Compression is enabled on a context basis. When enabled, all Streaming and Persistence related TCP-based communication into or out of the context is compressed by LZ4. A context with compression enabled will not accept source creation with transports other than TCP.

Subscribers will only successfully receive data if the receiver's context and the source's context share the same compression settings. A receiver created in a compression-enabled context will ignore topic resolution source advertisements for non-compressed sources, and will therefore not subscribe. Similarly, a receiver created in an non-compressed context will ignore topic resolution source advertisements for compressed sources. Topic resolution queries are also ignored by mismatched contexts. No warning will be logged when these topic resolution datagrams are ignored, but each time this happens, the context-level statistic tr_dgrams_dropped_type is incremented.

Compression is applied to unicast immediate messages as well, as invoked either directly by the user, or internally by functions like late join, request/response, and Persistence-related communication between sources, receivers, and Stores.

Brokered Queuing using AMQP does not use the UM compression feature. A UM brokered context does not allow compression to be enabled.

The compression-related configuration options used by the Ultra Messaging library are:

- **compression (context)**
- **tls_compression_negotiation_timeout (context)**

10.5.2 Compression and Persistence

Compression is designed to compress a data Transport Session. It is not intended to compress data at rest. When a persistent Store is used with the UM compression feature, the user messages are written to disk in uncompressed form.

10.5.3 Compression and Queuing

The UM compression feature does not apply to the AMQP connection to the brokered queue. UM does not currently support compression on the AMQP connection.

However, the ULB form of queuing does not use a broker. For ULB sources that are configured for TCP, the UM compression feature will compress the application data.

10.5.4 Compression and the DRO

When a DRO is used to route messages across Topic Resolution Domains (TRDs), be aware that the compression session is terminated at the DRO's proxy receiver/source. Because each endpoint portal on a DRO is implemented with its own context, care must be taken to ensure end-to-end compression (if desired). As a message is forwarded through a DRO network, it does not propagate the compression setting of the originator, so each portal needs to be appropriately compressed.

Possibly the most-useful application of the UM compression feature is not TCP sources, but rather DRO peer links. The compression feature is extended to DRO peer links, however peer links are not context-based and are not configured the same way. The following XML elements are used by the DRO to configure a peer link:

- **Router Element "<compression>"**

As with sources and receivers, the portals on both sides of a peer link must be configured for the same compression setting.

Notice that there is no DRO element corresponding to the context option **tls_compression_negotiation_timeout (context)**. The DRO peer link's negotiation timeout is hard-coded to 5 seconds.

See **DRO Configuration DTD** for details.

10.5.5 Compression and Encryption

See TLS and Compression.

10.5.6 Version Interoperability

It is not recommended to mix pre-6.9 contexts with compressed contexts on topics of shared interest. As mentioned above, if a compressed and an uncompressed context connect via TCP, the connection will fail and retry, resulting in flapping.

10.6 High-resolution Timestamps

This section introduces the use of high-resolution timestamps with LBT-RM.

The Ultra Messaging (UM) high-resolution message timestamp feature leverages the hardware timestamping function of certain Solarflare network interface cards (NICs) to measure sub-microsecond times that packets are transmitted and received. Solarflare's NICs and Onload kernel-bypass driver implement PTP to synchronize timestamps across the network, allowing very accurate one-way latency measurements. The UM timestamp feature requires Onload version 201509 or later.

For subscribers, each message's receive timestamp is delivered in the message's header structure (for C programs, **lbm_msg_t** field **hr_timestamp**, of type **lbm_timespec_t**). Each timestamp is a structure of 32 bits worth of seconds and 32 bits worth of nanoseconds. When both values are zero, the timestamp is not available.

For publishers, each message's transmit timestamp is delivered via the source event callback (for C programs, event type LBM_SRC_EVENT_TIMESTAMP). The same timestamp structure as above is delivered with the event, as well as the message's sequence number. Sending applications can be informed of the outgoing sequence number range of each message by using the extended form of the send function and supplying the LBM_SRC_SEND_EX_FL↩ AG_SEQUENCE_NUMBER_INFO flag. This causes the LBM_SRC_EVENT_SEQUENCE_NUMBER_INFO event to be delivered to the source event handler.

10.6.1 Timestamp Restrictions

Due to the specialized nature of this feature, there are several restrictions in its use.

- **Operating system**: Linux only. No timestamps will be delivered on other operating systems. Also, since the feature makes use of the rcvmmsg() function, no timestamps will be delivered on Linux kernels prior to 2.6.33 and glibc libraries prior to 2.12 (which was released in 2010).

- **Transport**: Source-based LBT-RM (multicast) Transport Sessions only. No timestamps will be delivered for MIM or other transport types.

- **Queuing**: Timestamps are not supported for broker-based queuing. If a ULB source is configured for LBT-↩ RM, send-side timestamps are not supported and will not be delivered if one or more receivers are registered. However, on the receive side, ULB messages are time stamped.

- **Loss**: If packet loss triggers LBT-RM's NAK/retransmit sequence, the send side will have multiple timestamps delivered, one for each multicast transmission. On the receive side, the timestamp of the first successfully received multicast datagram will be delivered.

- **Recovery**: For missed messages which are recovered via Late Join, Off-Transport Recovery (OTR), or the persistent Store, no timestamp will be delivered, either on the send side or the receive side.

- **Implicit batching**: If implicit batching is being used, only the first message in a batch will have a send-side timestamp delivered. When implicit batching is used, the sender must be prepared for some messages to not have timestamps delivered. On the receive side, all messages in a batch will have the same timestamp.

- **UM Fragmentation, send-side**: If user messages are too large to be contained in a single datagram, UM will fragment the message into multiple datagrams. On the send side, each datagram will trigger delivery of a timestamp.

  - UM Fragmentation, receive-side with **ordered_delivery (receiver)** set to 0 (arrival): Arrival-order delivery will result in each fragment being delivered separately, as it is received. Each fragment's message header will contain a timestamp. Arrival order delivery provides an accurate timestamp of when the complete message is received (although, as mentioned above, any fragment recovered via OTR or the persistent Store will not have a timestamp).

  - UM Fragmentation, receive-side with **ordered_delivery (receiver)** set to 1 or -1, (reassembly): Delivery with reassembly results in a single timestamp included in the message header. That timestamp corresponds to the arrival of the last fragment of the message (although, as mentioned above, any fragment recovered via OTR or the persistent Store will not have a timestamp). Note that this is not necessarily the last fragment received; if an intermediate datagram is lost and subsequently re-transmitted after a delay, that intermediate datagram will be the last one received, but its timestamp will not be used for the message. For example, if a three-fragment message is received in the order of F0, F2, F1, the timestamp for the message will correspond to F2, the last fragment of the message. If fragmented messages are being sent, and an accurate time of message completion is needed, arrival order delivery must be used.

- **UM Fragmentation plus implicit batching**: If user messages vary widely in size, some requiring fragmentation, and implicit batching is used be aware that a full fragment does not completely fill a datagram. For example, if a small message (less than 300 bytes) is sent followed by a large message requiring fragmentation, the first fragment of the large message will fit in the same datagram as the small message. In that case, on the send side, a timestamp will not be delivered for that first fragment. However, a timestamp will be delivered for the second fragment. On the receive side, the same restrictions apply as described with UM fragmentation.

- **Local loopback**: If an LBT-RM source and receiver share the same physical machine, the receive side will not have timestamps delivered.

10.6.2 Timestamp Configuration Summary

- **transport_lbtrm_source_timestamp (context)**

- **transport_lbtrm_receiver_timestamp (context)**

10.7 Unicast Immediate Messaging

Unicast Immediate Messaging (UIM) deviates from the normal publish/subscribe paradigm by allowing the publisher to send messages to a specific destination application context. Various features within UM make use of UIMs transparently to the application. For example, a persistent receiver sends consumption acknowledgements to the Store using UIM messages.

The application can make direct use of UIM in two ways:

- Calling a UIM send API function. See **lbm_unicast_immediate_message()** and **lbm_unicast_immediate↩ _request()**.

- Sending a response to a message using the Request/Response feature. See **lbm_send_response()**.

A UIM message can be associated with a topic string, but that topic is not used to determine where the message is sent to. Instead, the topic string is included in the message header, and the application must specify the destination of the desired context in the UIM send call.

Alternatively, an application can send a UIM message without a topic string. This so-called "topicless" message is not delivered to the receiving application via the normal receiver callback. Instead it is delivered by an optional context callback. See Receiving a UIM.

UIM messages are sent using the TCP protocol; no other protocol is supported for UIM. The TCP connection is created dynamically as-needed, when a message is sent. That is, when an application sends its first UIM message to a particular destination context, the sender's context holds the message and initiates the TCP connection. When the connection is accepted by the destination context, the sender sends the message. When the message is fully sent, the sender will keep the TCP connection open for a period of time in case more UIMs are sent to the same destination context. See UIM Connection Management.

10.7.1 UIM Reliability

Although UIM is implemented using TCP, it should not be considered "reliable" in the same way that UM sources are. There are a variety of ways that a UIM message can fail to be delivered to the destination. For example, if an overloaded DRO must forward the message to the destination TRD, the DRO might have to drop the message.

Note that the success or failure of delivery of a UIM cannot be determined by the sender. For example, calling **lbm_unicast_immediate_message()** or **lbm_send_response()** might return a successful status, but the message might have been queued internally for later transmission. So the success status only means that it was queued successfully. A subsequent failure may or may not trigger an error log; it depends on what the nature of the failure is.

As a result, applications are responsible for detecting failures (typically using timeouts) and implementing retry logic. If a higher degree of messaging reliability is required, normal UM sources should be used.

(For UIM messages that are sent transparently by UM features, various timeout/retry mechanisms are implemented internally.)

UIMs can also be lost as a result of TCP disconnections. See TCP Disconnections for more information.

10.7.2 UIM Addressing

There are three ways to specify the destination address of a UIM:

- Implicit – this method is used when a response is sent using the Request/Response feature. See **lbm_send_response()** for details.
- Explicit – this method uses a string of the form: "TCP:ip:port" (no DRO) or "TCP:domain:ip:port" (DRO in use). See **lbm_unicast_immediate_message()** for details.
- Source – this method uses a source string to send messages to the context which hosts a given source. See Sending to Sources for details.

In the Explicit addressing method, the "ip" and "port" refer to the binding of the destination context's UIM port (also called "request port"). By default, when a context is created, UM will select values from a range of possibilities for ip and port. However, this makes it difficult for a sender to construct an explicit address since the ip and port are not deterministic.

One solution is to explicitly set the ip and port for the context's UIM port using the configuration options: **request_tcp_port (context)** and **request_tcp_interface (context)**.

10.7.3 Receiving a UIM

There are two kinds of UIM messages:

- UIMs with a topic.

- UIMs with no topic (topicless).

To receive UIM messages with a topic, an application simply creates a normal receiver for that topic. Alternatively, it can create a wildcard receiver for a matching topic pattern. Finally, the application can also register a callback specifically for UIM messages that contain a topic but for which no matching receiver exists, using **immediate_message_topic_receiver_function (context)**. Alternatively, **lbm_context_rcv_immediate_topic_msgs()** can be used.

To receive UIM messages with no topic (topicless), the application must register a callback with the context for topicless messages, using **immediate_message_receiver_function (context)**). Alternatively, the API **lbm_context_rcv_immediate_msgs()** can be used to register that callback.

Note that only the specified destination context will deliver the message. If other applications have a receiver for that same topic, they will not receive a copy of that UIM message.

UIM Port

To receive UIMs, a context must bind to and listen on the "UIM Port", also known as the "Request Port". See UIM Addressing for details.

10.7.4 Sending a UIM

The following APIs are used to send application UIM messages:

- **lbm_unicast_immediate_message()**
- **lbm_unicast_immediate_request()**
- **lbm_send_response()** – see Request/Response for more information.
-

For the **lbm_unicast_immediate_message()** and **lbm_unicast_immediate_request()** APIs, the user has a choice between sending messages with a topic or without a topic (topicless). With the C API, passing a NULL pointer for the topic string sends a topicless message.

10.7.5 UIM Connection Management

The act of sending a UIM message will check to see if the context already has a TCP connection open to the destination context. If so, the existing connection is used to send the UIM. Otherwise, UM will initiate a new TCP connection to the destination context.

Once the message is sent, an activity deletion timer is started for the connection; see **response_tcp_deletion_timeout (context)**. If another UIM message is sent to the same destination, the activity deletion timer is canceled and restarted. Thus, if messages are sent periodically with a shorter period than the activity deletion timer, the TCP connection will remain established.

However, if no messages are sent for more time than the activity deletion timer, the timer will expire and the TCP connection will be deleted and resources cleaned up.

An exception to this workflow exists for the Request/Response feature. When a request message is received by a context, the context automatically initiates a connection to the requester, even though the application has not yet sent its response UIM. The activity deletion timer is not started at this time. When the application's receiver callback is invoked with the request message, the message contains a reference to a response object. This response object is used for sending response UIMs back to the requester. However, the act of sending these responses also does not start the activity deletion timer for the TCP connection. The activity deletion timer is only started when the response object is deleted (usually implicitly when the message itself is deleted, usually as a result of returning from the receiver callback).

Note that the application that receives a request has the option of retaining the message, which delays deletion of the message until the application explicitly decides to delete it. In this case, the TCP connection is held open for as long as the application retains the response object. When the application has finished sending any and all of its responses to the request and does not need the request object any more, it deletes the request object. This starts the activity deletion timer running.

Finally, note that there is a queue in front of the UIM connection which holds messages when the connection is slow. It is possible that messages are still held in the queue for transmission after the response object is deleted, and if the response message is very large and/or the destination application is slow processing responses, it is possible for data to still be queued when the activity deletion timer expires. In that case, UM does *not* delete the TCP connection, and instead restarts the activity deletion timer for the connection.

10.8 Multicast Immediate Messaging

Warning

> Multicast Immediate Messaging (MIM) is not recommended for general use. It is inefficient and can affect the operation of all applications on a UM network. This is partly because every message sent via the MIM protocol is distributed to *every* other application on the network, regardless of that application's interest in such messages.

MIM uses the same reliable multicast protocol as normal LBT-RM sources. MIM messages can be sent to a topic, in which case each receiving context will filter that message, discarding it if no receiver exists for that topic. MIM avoids using Topic Resolution by including the full topic string in the message, and sending it to a multicast group that all application contexts are configured to use.

Attention

> MIM reception is disabled by default. If you want an application to be able to receive MIM messages, you must set the configuration option **mim_address (context)** to the desired multicast group address.

A receiving context will receive the message and check to see if the application created a receiver for the topic. If so, then the message is delivered. However, if no receiver exists for that topic, the context checks to see if **immediate↩ _message_topic_receiver_function (context)** is configured. If so, then the message is delivered. But if neither exists, then the receiving context discards the message.

It is also possible to send a "topicless" message via MIM. The recipient context must have configured a topicless receiver using **immediate_message_receiver_function (context)**; otherwise the message is discarded.

A valid use case for MIM might be an application that starts running, sends a request message, gets back a response, and then exits. With this kind of short-lived application, it can be a burden to create a source and wait for it to resolve. With MIM, topic resolution is skipped, so no delay is needed prior to sending.

MIM is typically not used for normal Streaming data because messages are somewhat less efficiently handled than source-based messages. Inefficiencies derive from larger message sizes due to the inclusion of the topic name, and on the receiving side, the MIM Delivery Controller hashing of topic names to find receivers, which consumes some extra CPU. If you have a high-message-rate stream, you should use a source-based method and not MIM. If head-loss is a concern and delay before sending is not feasible, then consider using late join (although this replaces head-loss with some head latency).

Multicast Immediate Messaging can benefit from hardware acceleration. See **Transport Acceleration Options** for more information

Note

MIM is not compatible with Queuing, including ULB.

10.8.1 Temporary Transport Session

MIM uses the same reliable multicast algorithms as LBT-RM. When a publisher sends a message with **lbm_multicast_immediate_message()**, MIM creates a temporary Transport Session. Note that no topic-level source object is created.

MIM automatically deletes the temporary Transport Session after a period of inactivity defined by **mim_src_deletion_timeout (context)** which defaults to 30 seconds. A subsequent send creates a new Transport Session. Due to the possibility of head-loss in the switch, it is recommended that publisher use a long deletion timeout if they continue to use MIM after significant periods of inactivity.

MIM forces all topics across all publishers to be concentrated onto a single multicast address to which ALL applications listen, even if they aren't interested in any of the topics. Thus, all topic filtering must happen in UM.

MIM can also be used to send an UM request message with **lbm_multicast_immediate_request()**. For example, an application can use MIM to request initialization information right when it starts up. MIM sends the response directly to the initializing application, avoiding the topic resolution delay inherent in the normal source-based **lbm_send_request()** function.

10.8.2 MIM Notifications

MIM notifications differ in the following ways from normal UM source-based sending.

- When a publisher's MIM Transport Session times out and is deleted, the receiving applications do not receive an EOS notification.
- Applications with a source notification callback (**resolver_source_notification_function (context)**) are not informed of a MIM sender. This is because source notification is based on Topic Resolution, and MIM does not use it.
- MIM sending supports the non-blocking flag. However, it does not provide an **LBM_SRC_EVENT_WAKEUP** notification when the MIM session becomes writable again.
- MIM sends unrecoverable loss notifications to a context callback, not to a receiver callback. See MIM Loss Handling.

10.8.3 Receiving Immediate Messages

To receive MIM messages with a topic, an application simply creates a normal receiver for that topic. Alternatively, it can create a wildcard receiver for a matching topic pattern. Finally, the application can also register a callback specifically for UIM messages that contain a topic but for which no matching receiver exists, using **immediate_message_topic_receiver_function (context)**. Alternatively, **lbm_context_rcv_immediate_topic_msgs()** can be used.

To receive MIM messages with no topic (topicless), the application must register a callback for topicless messages, using **immediate_message_receiver_function (context)**). Alternatively, **lbm_context_rcv_immediate_msgs()** can be used.

If needed, an application can send topicless messages using MIM. A MIM sender passes in a NULL string instead of a topic name. The message goes out on the MIM multicast address and is received by all other receivers. A receiving application can use **lbm_context_rcv_immediate_msgs()** to set the callback procedure and delivery method for topicless immediate messages.

10.8.4 MIM and Wildcard Receivers

When an application receives an immediate message, it's topic is hashed to see if there is at least one regular (non-wildcard) receiver object listening to the topic. If so, then MIM delivers the message data to the list of receivers.

However, if there are no regular receivers for that topic in the receive hash, MIM runs the message topic through all existing wildcard patterns and delivers matches to the appropriate wildcard receiver objects without creating sub-receivers. The next MIM message received for the same topic will again be run through all existing wildcard patterns. This can consume significant CPU resources since it is done on a per-message basis.

10.8.5 MIM Loss Handling

The receiving application can set up a context callback to be notified of MIM unrecoverable loss (**lbm_mim_↩ unrecloss_function_cb()**). It is not possible to do this notification on a topic basis because the receiving UM has no way of knowing which topics were affected by the loss.

10.8.6 MIM Configuration

- As of UM 3.1, MIM supports ordered delivery.
- As of UM 3.3.2, the MIM configuration option, **mim_ordered_delivery (context)** defaults to ordered delivery.
- **IMPORTANT:** As of UM 6.8, MIM reception is disabled by default. If you want an application to be able to receive MIM messages, you must set the configuration option **mim_address (context)** to the desired multicast group address.

The configuration options for MIM are described here:

- **Multicast Immediate Messaging Network Options**
- **Multicast Immediate Messaging Reliability Options**
- **Multicast Immediate Messaging Operation Options**

10.8.7 MIM Example Applications

UM includes two example applications that illustrate MIM.

- **Example lbmimsg.c** - application that sends immediate messages as fast as it can to a given topic (single source). See also the Java example, **Example lbmimsg.java**, and the .NET example, **Example lbmimsg.cs**.

- **Example lbmireq.c** - application that sends immediate requests to a given topic (single source) and waits for responses.

10.9 HyperTopics

Warning

> **The HyperTopics feature is deprecated and may be removed in a future release**. Users should use normal wildcard receivers with "PCRE" regular expression pattern matching. See UM Wildcard Receivers.

For information regarding HyperTopics, see **C HyperTopics Details**.

10.10 Application Headers

The Application Headers feature is an older method of adding untyped, unstructured metadata to messages.

Warning

> The Application Headers feature is deprecated and may be removed in a future version. Informatica recommends using Message Properties.

10.10.1 Application Headers Usage

Send Message with Application Headers

To send a message with one or more application headers attached, follow these steps:

1. Create an application header chain object using **lbm_apphdr_chain_create()**.

```
lbm_apphdr_chain_t *chain_obj;
...
err = lbm_apphdr_chain_create(&chain_obj);
```

2. Declare a chain element structure **lbm_apphdr_chain_elem_t**.

```
lbm_apphdr_chain_elem_t chain_el;
```

3. Set **lbm_apphdr_chain_elem_t_stct::type** to LBM_CHAIN_ELEM_USER_DATA.

4. Set **lbm_apphdr_chain_elem_t_stct::subtype** to a desired integer.

5. Set **lbm_apphdr_chain_elem_t_stct::data** to point to the desired bytes of data.

6. Set **lbm_apphdr_chain_elem_t_stct::len** to the number of bytes of data.

```
chain_el.type = LBM_CHAIN_ELEM_USER_DATA;
chain_el.subtype = MY_SUBTYPE;
chain_el.data = "abc";
chain_el.len = 4;
```

7. Declare an extended send information structure **lbm_src_send_ex_info_t**.

   ```
   lbm_src_send_ex_info_t ex_info;
   ```

8. Set the **LBM_SRC_SEND_EX_FLAG_APPHDR_CHAIN** bit in the **lbm_src_send_ex_info_t_stct::flags** field.

9. Set the **lbm_src_send_ex_info_t_stct::apphdr_chain** field with the application header chain object created in step 1.

   ```
   memset(&ex_info, 0, sizeof(ex_info));
   ex_info.flags = LBM_SRC_SEND_EX_FLAG_APPHDR_CHAIN;
   ex_info.apphdr_chain = chain_obj;
   ```

10. Call **lbm_src_send_ex()** passing the extended send information structure.

    ```
    err = lbm_src_send_ex(src, my_buffer, my_len, LBM_MSG_FLUSH, &ex_info)
    ```

11. Delete the application header chain object using **lbm_apphdr_chain_delete()**.

    ```
    err = lbm_apphdr_chain_delete(chain_obj);
    ```

Instead of creating and deleting the application header chain with each message, it may be retained and re-used. However, it may not be modified.

Receive Message with Application Headers

To handle a received message that may contain application headers, follow these steps:

1. In the receiver callback, create an application header iterator using **lbm_apphdr_chain_iter_create_from_msg()**. If it returns LBM_FAILURE then the message has no application headers.

   ```
   lbm_apphdr_chain_iter_t *apphdr_itr;
   ...
   err = lbm_apphdr_chain_iter_create_from_msg(&apphdr_itr, msg);
   if (err == LBM_OK) {  /* App header exists. */
   ```

2. If **lbm_apphdr_chain_iter_create_from_msg()** returns LBM_OK, declare a chain element structure **lbm_apphdr_chain_elem_t** pointer and set it using **lbm_apphdr_chain_iter_current()**.

   ```
   do {  /* Loop through each app header. */
     lbm_apphdr_chain_elem_t *chain_el;
     chain_el = lbm_apphdr_chain_iter_current(&apphdr_itr);
   ```

3. Access the chain element subtype through the chain element structure using **lbm_apphdr_chain_elem_t_stct::subtype**.

4. Access the chain element length through the chain element structure using **lbm_apphdr_chain_elem_t_stct::len**.

5. Access the chain element data through the chain element structure using **lbm_apphdr_chain_elem_t_stct::data**.

   ```
   printf("  chain_el.type=%d, chain_el.len=%lu, chain_el.data='%s'\n",
          chain_el->type, chain_el->len, (char *)chain_el->data);
   ```

6. To step to the next application header (if any), call **lbm_apphdr_chain_iter_next()**. This function returns LBM_OK if there is another application header; go to step 3.

   ```
     err = lbm_apphdr_chain_iter_next(&apphdr_itr);
   while (err == LBM_OK);
   ```

7. If there are no more application headers, it returns LBM_FAILURE.

8. Delete the iterator using **lbm_apphdr_chain_iter_delete()**.

```
err = lbm_apphdr_chain_iter_delete(apphdr_itr);
```

Application Headers are not compatible with the following UM features:

- Smart Sources.
- Request/Response
- Unicast Immediate Messaging
- Multicast Immediate Messaging

Note that the user data provided for the application header is not typed. UM cannot reformat messages for architectural differences in a heterogeneous environment. For example, UM does not do byte-swapping between big and little endian CPUs.

10.11 Message Properties

The Message Properties feature allows your application to add typed metadata to messages as name/value pairs. UM allows eight property types: boolean, byte, short, int, long, float, double, and string. See Message Properties Data Types.

With the UMQ product, the UM message property object supports the standard JMS message properties specification.

Message properties are not compatible with the following UM features:

- Arrival Order, Fragments Not Reassembled (ordered_delivery 0).
- Transport LBT-SMX.

10.11.1 Message Properties Usage

Send Message with Message Properties

For sending messages with message properties using Smart Sources, see Smart Sources and Message Properties.

To send a message with one or more message properties attached using normal sources (i.e. not Smart Sources), follow these steps:

1. Create a message properties object using **lbm_msg_properties_create()**.

```
lbm_msg_properties_t *prop_obj;
...
err = lbm_msg_properties_create(&prop_obj);
```

2. Add one or more properties using **lbm_msg_properties_set()**.

```
lbm_uint32_t int_prop_val = MY_VAL;
...
err = lbm_msg_properties_set(prop_obj, "My Prop 1", &int_prop_val,
                             LBM_MSG_PROPERTY_INT, sizeof(int_prop_val));
```

3. Declare an extended send information structure **lbm_src_send_ex_info_t**.

```
lbm_src_send_ex_info_t ex_info;
```

4. Set the **LBM_SRC_SEND_EX_FLAG_PROPERTIES** bit in the **lbm_src_send_ex_info_t_stct::flags** field.

5. Set the **lbm_src_send_ex_info_t_stct::properties** field with the properties object created in step 1.

```
memset(&ex_info, 0, sizeof(ex_info));
ex_info.flags = LBM_SRC_SEND_EX_FLAG_PROPERTIES;
ex_info.properties = prop_obj;
```

6. Call **lbm_src_send_ex()** passing the extended send information structure.

```
err = lbm_src_send_ex(src, my_buffer, my_len, LBM_MSG_FLUSH, &ex_info);
```

7. Delete the message properties object using **lbm_msg_properties_delete()**.

```
err = lbm_msg_properties_delete(prop_obj);
```

Instead of creating and deleting the properties object with each message, it may be retained and re-used. It can also be modified using **lbm_msg_properties_clear()** and **lbm_msg_properties_set()**.

Receive Message with Message Properties

To handle a received message that may contain message properties, follow these steps:

1. In the receiver callback, check the message's properties field **lbm_msg_t_stct::properties**. If it is NULL, the message has no properties.

```
if (msg->properties != NULL) {
  /* Handle message properties. */
```

2. If **lbm_msg_t_stct::properties** is non-null, create a property iterator object using **lbm_msg_properties_iter_create()**.

```
lbm_msg_properties_iter_t *prop_itr;
...
err = lbm_msg_properties_iter_create(&prop_itr);
```

3. Set the iterator to the first property in the message using **lbm_msg_properties_iter_first()**.

```
err = lbm_msg_properties_iter_first(prop_itr, msg->properties);
do {  /* Loop through the properties. */
```

4. Access that property's name through the iterator using **lbm_msg_properties_iter_t_stct::name**.

5. Access that property's type through the iterator using **lbm_msg_properties_iter_t_stct::type**.

6. Access that property's value through the iterator using **lbm_msg_properties_iter_t_stct::data** (must be cast to the appropriate data type).

```
if (prop_itr->type == LBM_MSG_PROPERTY_INT) {
  printf("  prop_itr.data=%d\n", *((lbm_uint32_t *)(prop_itr->data)));
```

7. To step to the next property (if any), call **lbm_msg_properties_iter_next()**. This function returns LBM_OK if there is another property; go to step 4. If there are no more properties, it returns LBM_FAILURE.

```
  err = lbm_msg_properties_iter_next(prop_itr);
} while (err == LBM_OK);
```

8. Delete the iterator using **lbm_msg_properties_iter_delete()**.

```
err = lbm_msg_properties_iter_delete(prop_itr);
```

Instead of iterating through the properties, it is also possible to access properties by name using **lbm_msg_properties_get()**. However, this can be less efficient.

For a C example on how to iterate received message properties, see **Example lbmrcv.c**.

Note that if a publisher sends a message with no properties set, best practice is to not pass the property object to the send function. It is permissible to pass a property object with no properties set, but it adds latency and overhead. Also, it causes the receiver to get a non-null msg->properties field. When the receiver calls **lbm_msg_properties_iter_first()**, it will return an error because there isn't a "first" property. It is better for the publisher to not pass an empty properties object so that the receiver will get a NULL msg->properties field.

10.11.2 Message Properties Data Types

Due to differences in the integer variable sizes on different compilers on different platforms, Informatica recommends using the following C data types for the corresponding message property data types:

Property Type	C Type
LBM_MSG_PROPERTY_BOOLEAN	char
LBM_MSG_PROPERTY_BYTE	char
LBM_MSG_PROPERTY_SHORT	**lbm_uint16_t**
LBM_MSG_PROPERTY_INT	**lbm_int32_t**
LBM_MSG_PROPERTY_LONG	**lbm_int64_t**
LBM_MSG_PROPERTY_FLOAT	float
LBM_MSG_PROPERTY_DOUBLE	double
LBM_MSG_PROPERTY_STRING	char array

10.11.3 Message Properties Performance Considerations

Ultra Messaging sends property names on the wire with every message. To reduce bandwidth requirements, minimize the length and number of properties. When coding sources, consider the following sequence of guidelines:

1. Allocate a data structure to store message properties objects. This can be a thread-local structure if you use a relatively small number of threads, or a thread-safe pool of objects.

2. Before sending, retrieve a message properties object from the pool. If an object is not available, create a new object.

3. Set properties for the message.

4. Send the message using the appropriate API call, passing in the properties object.

5. After the send completes, clear the message properties object and return it to the pool.

When coding receivers in Java or .NET, call Dispose() on messages before returning from the application callback. This allows Ultra Messaging to internally recycle objects, and limits object allocation.

10.11.4 Smart Sources and Message Properties

Smart Sources support a limited form of message properties. Only 32-bit integer property types are allowed with Smart Sources. Also, property names are limited to 7 ASCII characters. Finally, the normal message properties object **lbm_msg_properties_t** and its APIs *are not used* on the sending side. Rather a streamlined method of specifying message properties for sending is used.

As with most of Smart Source's internal design, the message header for message properties must be pre-allocated with the maximum number of desired message properties. This is done at creation time for the Smart Source using the configuration option **smart_src_message_property_int_count (source)**.

Sending messages with message properties must be done using the **lbm_ssrc_send_ex()** API, passing it the desired properties with the **lbm_ssrc_send_ex_info_t** structure. The first call to send with message properties will serialize the supplied properties and encode them into the pre-allocated message header.

Subsequent calls to send with message properties will ignore the passed-in properties and simply re-send the previously-serialized header.

If an application needs to change the message property values after that initial send, the "update" flag flag can be used, which will trigger modification of the property values. This "update" flag cannot be used to change the number of properties, or the key names of the properties.

If an application needs messages with different numbers of properties and/or different key names of properties, the most efficient way to accomplish this is with multiple message buffers. Each buffer should be associated with a desired set of properties. When a message needs to be sent, the proper buffer is selected for building the message. This avoid the overhead of serializing the properties with each send call.

However, if the application requires dynamic construction of properties, a single buffer can be used along with the "rebuild" flag to trigger a full serialization of the properties.

Note

> If using both message properties and Spectrum with a single Smart Source, there is an added restriction: it is not possible to send a message omitting only one of those features. I.e. if both are enabled when the Smart Source is created, it is not possible to send a message with a message property and not a channel, and it is not possible to send a message with a channel and not a property. This is because the message header is defined at Smart Source creation, and the header either must contain both or neither.

10.11.5 Smart Source Message Properties Usage

For a full example of message property usage with Smart Source, see **Example lbmssrc.c** or **Example lbmssrc.java**.

The first message with a message property sent to a Smart Source follows a specific sequence:

1. Create the topic object with the configuration option **smart_src_message_property_int_count (source)** set to the maximum number of properties desired on a message.

2. Create the Smart Source with **lbm_ssrc_create()**.

3. Allocate one or more message buffers with **lbm_ssrc_buff_get()**. You might allocate one for messages that have properties, and another for messages that don't.

4. When preparing the first message with message properties to be sent, define the properties using a **lbm_ssrc_send_ex_info_t** structure:

```
char *prop_name_array[3]; /* Array of property names. */
prop_name_array[0] = "abc"; /* 7 ascii characters or less. */
prop_name_array[1] = "XYZ";
prop_name_array[2] = "123";
```

```
lbm_int32_t prop_value_array[3]; /* Array of property values. */
prop_value_array[0] = 29;
prop_value_array[1] = -300;
prop_value_array[2] = 0;

lbm_ssrc_send_ex_info_t ss_send_info;
memset((char *)&ss_send_info, 0, sizeof(ss_send_info));
ss_send_info.mprop_int_cnt = 3;
ss_send_info.mprop_int_keys = prop_name_array;
ss_send_info.mprop_int_vals = prop_value_array;
```

5. Send the message using **lbm_ssrc_send_ex()** and the **LBM_SSRC_SEND_EX_FLAG_PROPERTIES** flag:

   ```
   ss_send_info.flags = LBM_SSRC_SEND_EX_FLAG_PROPERTIES;
   err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
   ```

   Since this is the first send with message properties, UM will serialize the properties and set up the message header. (It is not valid to set the **LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES** flag on this first send with message properties.)

For subsequent sends, there are different use cases:

- Send the message with the same properties and values. You can re-use the same message buffer and **lbm_ssrc_send_ex_info_t** structure:

  ```
  /* The ss_send_info.flags still has LBM_SSRC_SEND_EX_FLAG_PROPERTIES set */
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
  ```

- Send with message properties after having made changes to the property values (but not the keys or the number of properties) by setting the **LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES** flag:

  ```
  prop_value_array[0] = 28;  /* Change property value. */
  ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES;
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
  ss_send_info.flags &= ~LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES;
  ```

- Send a message with either a different number of properties, and/or different key names by setting the **LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER** flag:

  ```
  /* Send with only the first 2 properties. */
  ss_send_info.mprop_int_cnt = 2;
  ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
  ss_send_info.flags &= ~ LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
  ```

- Send a message without any message properties. This is a subset of the previous case (changing the number of properties). Use the **LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER** flag and clear the **LBM_SSRC_SEND_EX_FLAG_PROPERTIES** flag:

  ```
  /* Clear the properties flag so no properties will be sent. */
  ss_send_info.flags &= ~ LBM_SSRC_SEND_EX_FLAG_PROPERTIES;
  ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
  ss_send_info.flags &= ~ LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
  ```

To be more efficient, instead of using the **LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER** flag, you can use different message buffers (allocated with **lbm_ssrc_buff_get()**) for each message property structure. This saves the time required to re-serialize the message properties each time you want to use a different property structure.

10.12 Request/Response Model

Request/response is a very common messaging model whereby a client sends a "request" message to a server and expects a response. The server processes the request and return a response message to the originating client.

The UM request/response feature simplifies implementation of this model in the following ways:

- Handling the request's "return address", eliminating the need for the client to create an artificial guaranteed-unique topic for the response.
- Establishing a linkage between a request and its response(s), allowing multiple requests to be outstanding, and associating each response message with its corresponding request message.
- Supporting multiple responses per request, both by allowing multiple servers to receive the request and each one responding, and by allowing a given server to respond with multiple messages.

Attention

> Although the response messages are sent by TCP, there are still failure scenarios where a response can be lost. The use of DROs can make it even more likely. Informatica recommends applications that use request/response implement timeout/retry logic. See UIM Reliability.

10.12.1 Request Message

UM provides three ways to send a request message.

- **lbm_send_request()** to send a request to a topic via a source object. Uses the standard source-based transports (TCP, LBT-RM, LBT-RU).
- **lbm_multicast_immediate_request()** to send a request to a topic as a multicast immediate message. See Multicast Immediate Messaging.
- **lbm_unicast_immediate_request()** to send a request to a topic as a unicast immediate message.

When the client application sends a request message, it references an application callback function for responses and a client data pointer for application state. The send call returns a "request object". As one or more responses are returned, the callback is invoked to deliver the response messages, associated with the request's client data pointer. The requesting application decides when its request is satisfied (perhaps by completeness of a response, or by timeout), and it calls **lbm_request_delete()** to delete the request object. Even if the server chooses to send additional responses, they will not be delivered to the requesting application after it has deleted the corresponding request object.

10.12.2 Response Message

The server application receives a request via the normal message receive mechanism, but the message is identified as type "request". Contained within that request message's header is a response object, which serves as a return address to the requester. The server application responds to an UM request message by calling **lbm_send_response()**. The response message is sent unicast via a dynamic TCP connection managed by UM.

Warning

> The **lbm_send_response()** function may not be called from a context thread callback. If the application needs to send the response from the receiver callback (or any other UM callback), it must associate that callback with an event queue.

10.12.3 Response Size

A UM response message can be of any arbitrary size. However, given that responses are sent as UIMs, and UIMs are not guaranteed, UM responses can be lost. This is especially true if DROs are being used. Larger UM responses have a higher probability of being lost than smaller UM responses. In particular, UM responses larger than 65,000 bytes can be lost even if network packets are simply delivered out of order, as might happen if the DROs are configured to use a **UDP Peer Link**.

This does not mean that you must avoid sending large response messages. Rather it emphasizes the importance of using an application-level timeout/retry algorithm for request/response. In practice, even multi-megabyte responses would be lost only rarely, but it can happen.

10.12.4 Response Deletion

Since the response object is part of the message header, it is normally deleted at the same time that the message is deleted, which typically happens automatically when the receiver callback returns. However, there are times when the application needs the scope of the response object to extend beyond the execution of the receiver callback. One method of extending the lifetime of the response object is to "retain" the request message, using **lbm_msg_retain()**.

However, there are times when the size of the request message makes retention of the entire message undesirable. In those cases, the response object itself can be extracted and retained separately by saving a copy of the response object pointer and setting the message header's response pointer to NULL (to prevent UM from deleting the response object when the message is deleted).

There are even occasions when an application needs to transfer the responsibility of responding to a request message to a different process entirely. I.e. the server which receives the request is not itself able to respond, and needs to send a message (not necessarily the original request message) to a different server. In that case, the first server which receives the request must serialize the response object to type **lbm_serialized_response_t** by calling **lbm_serialize_response()**. It includes the serialized response object in the message forwarded to the second server. That server de-serializes the response object by calling **lbm_deserialize_response()**, allowing it to send a response message to the original requesting client.

10.12.5 TCP Management

UM creates and manages the special TCP connections for responses, maintaining a list of active response connections. When an application sends a response, UM scans that list for an active connection to the destination. If it doesn't find a connection for the response, it creates a new connection and adds it to the list. After the **lbm_send_response()** function returns, UM schedules the **response_tcp_deletion_timeout (context)**, which defaults to 2 seconds. If a second request comes in from the same application before the timer expires, the responding application simply uses the existing connection and restarts the deletion timer.

It is conceivable that a very large response could take more than the **response_tcp_deletion_timeout (context)** default (2 seconds) to send to a slow-running receiver. In this case, UM automatically increases the deletion timer as needed to ensure the last message completes.

10.12.6 Request/Response Configuration

See the UM Configuration Guide for the descriptions of the Request/Response configuration options:

- **Unicast Immediate Messaging Network Options**
- **Unicast Immediate Messaging Operation Options**

Note

> If your application is running within an UM context where the configuration option, **request_tcp_bind_request_port (context)** has been set to zero, UIM port binding has been turned off, which also disables the Request/Response feature.

10.12.7 Request/Response Example Applications

UM includes two example applications that illustrate Request/Response.

- **Example lbmreq.c** - application that sends requests on a given topic (single source) and waits for responses. See also the Java example, **Example lbmreq.java** and the .NET example **Example lbmreq.cs**.
- **Example lbmresp.c** - application that waits for requests and sends responses back on a given topic (single receiver). See also the Java example, **Example lbmresp.java** and the .NET example **Example lbmresp.cs**.

We can demonstrate a series of 5 requests and responses with the following procedure:

- Run **lbmresp -v topicname**
- Run **lbmreq -R 5 -v topicname**

LBMREQ

Output for lbmreq should resemble the following:

```
$ lbmreq -R 5 -q topicname
Event queue in use
Using TCP port 4392 for responses
Delaying requests for 1000 milliseconds
Sending request 0
Starting event pump for 5 seconds.
Receiver connect [TCP:10.29.1.78:4958]
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 1
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 2
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 3
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 4
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
Quitting...
```

LBMRESP

Output for lbmresp should resemble the following:

```
$ lbmresp -v topicname
Request [topicname][TCP:10.29.1.78:14371][0], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][1], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][2], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][3], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][4], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
[topicname][TCP:10.29.1.78:14371], End of Transport Session
```

10.13 Self Describing Messaging

The UM Self-Describing Messaging (SDM) feature provides an API that simplifies the creation and use of messages by your applications. An SDM message contains one or more fields and each field consists of the following:

- A name
- A type
- A value

Each named field may appear only once in a message. If multiple fields of the same name and type are needed, array fields are available. A field in a nested message may have the same name as a field in the outer message.

SDM is particularly helpful for creating messages sent across platforms by simplifying the creation of data formats. SDM automatically performs platform-specific data translations, eliminating endian conflicts.

Using SDM also simplifies message maintenance because the message format or structure can be independent of the source and receiver applications. For example, if your receivers query SDM messages for particular fields and ignore the order of the fields within the message, a source can change the field order if necessary with no modification of the receivers needed.

See the `C`, `Java`, and `.NET` API references for details.

Informatica generally recommends the use of Pre-Defined Messages, which is more efficient than self-describing messages.

10.14 Pre-Defined Messages

The UM Pre-Defined Messages (PDM) feature provides an API similar to the SDM API, but allows you to define messages once and then use the definition to create messages that may contain self-describing data. Eliminating the need to repeatedly send a message definition increases the speed of PDM over SDM. The ability to use arrays created in a different programming language also improves performance.

The PDM library lets you create, serialize, and deserialize messages using pre-defined knowledge about the possible fields that may be used. You can create a definition that a) describes the fields to be sent and received in a message, b) creates the corresponding message, and c) adds field values to the message. This approach offers several performance advantages over SDM, as the definition is known in advance. However, the usage pattern is slightly different from the SDM library, where fields are added directly to a message without any type of definition.

A PDM message contains one or more fields and each field consists of the following:

- A name
- A type
- A value

Each named field may appear only once in a message. If multiple fields of the same name and type are needed, array fields are available. A field in a nested message may have the same name as a field in the outer message.

See the `C`, `Java`, and `.NET` API references for details.

The C API also has information and code samples about how to create definitions and messages, set field values in a message, set the value of array fields in a message, serialize, deserialize and dispose of messages, and fetch values from a message.

See **C PDM Details**.

10.14.1 Typical PDM Usage Patterns

The typical PDM usage patterns can usually be broken down into two categories: sources (which need to serialize a message for sending) and receivers (which need to deserialize a message to extract field values). However, for optimum performance for both sources and receivers, first set up the definition and a single instance of the message only once during a setup or initialization phase, as in the following example workflow:

1. Create a definition and set its id and version.
2. Add field information to the definition to describe the types of fields to be in the message.
3. Create a single instance of a message based on the definition.
4. Set up a source to do the following:
   - Add field values to the message instance.
   - Serialize the message so that it can be sent.
5. Likewise, set up a receiver to do the following:
   - Deserialize the received bytes into the message instance.
   - Extract the field values from the message.

10.14.2 Getting Started with PDM

PDM APIs are provided in C, Java, and C#, however, the examples in this section are Java based.

PDM Code Example, Source

Translating the Typical PDM Usage Patterns to Java for a source produces the following:

```
private PDMDefinition defn;
private PDMMessage msg;
private PDMFieldInfo fldInfo100;
private PDMFieldInfo fldInfo101;
private PDMFieldInfo fldInfo102;

public void setupPDM() {
  //Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);

  //Set the definition id and version
  defn.setId(1001);
  defn.setMsgVersMajor((byte)1);
  defn.setMsgVersMinor((byte)0);

  //Create information for a boolean, int32, and float fields (all required)
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);

  //Finalize the definition and create the message defn.finalizeDef();
  msg = new PDMMessage(defn);
}

public void sourceUsePDM() {
  //Call the function to setup the definition and message
  setupPDM();

  //Example values for the message boolean
  fld100Val = true;
  int fld101Val = 7;
  float fld102Val = 3.14F;

  //Set each field value in the message
  msg.setFieldValue(fldInfo100, fld100Val);
  msg.setFieldValue(fldInfo101, fld101Val);
  msg.setFieldValue(fldInfo102, fld102Val);

  //Serialize the message to bytes
  byte[] buffer = msg.toBytes();
}
```

PDM Code Example, Receiver

Translating the Typical PDM Usage Patterns to Java for a receiver produces the following:

```
private PDMDefinition defn;
private PDMMessage msg;
private PDMFieldInfo fldInfo100;
private PDMFieldInfo fldInfo101;
private PDMFieldInfo fldInfo102;

public void setupPDM() {
  //Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);

  //Set the definition id and version
  defn.setId(1001);
  defn.setMsgVersMajor((byte)1);
  defn.setMsgVersMinor((byte)0);

  //Create information for a boolean, int32, and float field (all required)
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);

  //Finalize the definition and create the message
  defn.finalizeDef();
  msg = new PDMMessage(defn);
}

public void receiverUsePDM(byte[] buffer) {
  //Call the function to setup the definition and message
  setupPDM();

  //Values to be retrieved from the message
  boolean fld100Val;
  int fld101Val;
  float fld102Val;

  //Deserialize the bytes into a message
  msg.parse(buffer);

  //Get each field value from the message
  fld100Val = msg.getFieldValueAsBoolean(fldInfo100);
  fld101Val = msg.getFieldValueAsInt32(fldInfo101);
  fld102Val = msg.getFieldValueAsFloat(fldInfo102);
}
```

PDM Code Example Notes

In the examples above, the setupPDM() function is called once to set up the PDM definition and message. It is identical in both the source and receiver cases and simply sets up a definition that contains three required fields with integer names (100, 101, 102). Once finalized, it can create a message that leverages its pre-defined knowledge about these three required fields. The source example adds the three sample field values (a boolean, int32, and float) to the message, which is then serialized to a byte array. In the receiver example, the message parses a byte array into the message and then extracts the three field values.

10.14.3 Using the PDM API

The following code snippets expand upon the previous examples to demonstrate the usage of additional PDM functionality (but use "..." to eliminate redundant code).

Reusing the Message Object

Although the examples use a single message object (which provides performance benefits due to reduced message creation and garbage collection), it is not explicitly required to reuse a single instance. However, multiple threads should not access a single message instance.

Number of Fields

Although the number of fields above is initially set to 3 in the PDMDefinition constructor, if you add more fields to the definition with the addFieldInfo method, the definition grows to accommodate each field. Once the definition is finalized, you cannot add additional field information because the definition is now locked and ready for use in a message.

String Field Names

The examples above use integer field names in the setupPDM() function when creating the definition. You can also use string field names when setting up the definition. However, you still must use a FieldInfo object to set or get a field value from a message, regardless of field name type. Notice that false is passed to the PDMDefinition constructor to indicate string field names should be used. Also, the overloaded addFieldInfo function uses string field names (.Field100.) instead of the integer field names.

```
...
public void setupPDM() {
  //Create the definition with 3 fields and using string field names
  defn = new PDMDefinition(3, false);
  ...
  //Create information for a boolean, int32, and float field (all required)
  fldInfo100 = defn.addFieldInfo("Field100", PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo("Field101", PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo("Field102", PDMFieldType.FLOAT, true);
  ...
}
...
```

Retrieving FieldInfo from the Definition

At times, it may be easier to lookup the FieldInfo from the definition using the integer name (or string name if used). This eliminates the need to store the reference to the FieldInfo when getting or setting a field value in a message, but it does incur a performance penalty due to the lookup in the definition to retrieve the FieldInfo. Notice that there are no longer FieldInfo objects being used when calling addFieldInfo and a lookup is being done for each call to msg.getFieldValueAs∗ to retrieve the FieldInfo by integer name.

```
private PDMDefinition defn;
private PDMMessage msg;

public void setupPDM() {
  ...
  //Create information for a boolean, int32, and float field (all required)
  defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  defn.addFieldInfo(101, PDMFieldType.INT32, true);
  defn.addFieldInfo(102, PDMFieldType.FLOAT, true);
  ...
}

public void receiverUsePDM(byte[] buffer) {
  ...
  //Get each field value from the message
  fld100Val = msg.getFieldValueAsBoolean(defn.getFieldInfo(100));
  fld101Val = msg.getFieldValueAsInt32(defn.getFieldInfo(101));
  fld102Val = msg.getFieldValueAsFloat(defn.getFieldInfo(102));
}
```

Required and Optional Fields

When adding field information to a definition, you can indicate that the field is optional and may not be set for every message that uses the definition. Do this by passing false as the third parameter to the addFieldInfo function. Using required fields (fixed-required fields specifically) produces the best performance when serializing and deserializing messages, but causes an exception if all required fields are not set before serializing the message. Optional fields allow the concept of sending "null" as a value for a field by simply not setting that field value on the source side before serializing the message. However, after parsing a message, a receiver should check the isFieldValueSet function for an optional field before attempting to read the value from the field to avoid the exception mentioned above.

```
...
private PDMFieldInfo fldInfo103;
...
public void setupPDM() {
  ...
  //Create information for a boolean, int32, and float field (all required)
  // as well as an optional int8 field
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);
  fldInfo103 = defn.addFieldInfo(103, PDMFieldType.INT8, false);
  ...
}

public void sourceUsePDM() {
  ...
  //Set each field value in the message
  // except do not set the optional field
  msg.setFieldValue(fldInfo100, fld100Val);
  msg.setFieldValue(fldInfo101, fld101Val);
  msg.setFieldValue(fldInfo102, fld102Val);
  ...
}

...
private PDMFieldInfo fldInfo103;
...
public void setupPDM() {
  ...
  //Create information for a boolean, int32, and float field (all required)
  // as well as an optional int8 field
  fldInfo103 = defn.addFieldInfo(103, PDMFieldType.INT8, false);
  ...
}

public void receiverUsePDM(byte[] buffer) {
  ...
  byte fld103Val;
  ...
  if(msg.isFieldValueSet(fldInfo103)) {
    fld103Val = msg.getFieldValueAsInt8(fldInfo103);
  }
}
```

Fixed String and Fixed Unicode Field Types

A variable length string typically does not have the performance optimizations of fixed-required fields. However, by indicating "required", as well as the field type FIX_STRING or FIX_UNICODE and specifying an integer number of fixed characters, PDM sets aside an appropriate fixed amount of space in the message for that field and treats it as an optimized fixed-required field. Strings of a smaller length can still be set as the value for the field, but the message allocates the specified fixed number of bytes for the string. Specify Unicode strings in the same manner (with FIX_UNICODE as the type) and in "UTF-8" format.

```
...
private PDMFieldInfo fldInfo104;
...
public void setupPDM() {
  ...
  fldInfo104 = defn.addFieldInfo(104, PDMFieldType.FIX_STRING, 12, true);
  ...
}

public void sourceUsePDM() {
  ...
  String fld104Val = "Hello World!";

  //Set each field value in the message
  // except do not set the optional field
  msg.setFieldValue(fldInfo100, fld100Val);
  msg.setFieldValue(fldInfo101, fld101Val);
  msg.setFieldValue(fldInfo102, fld102Val);
  msg.setFieldValue(fldInfo104, fld104Val);
  ...
}
```

```
...
private PDMFieldInfo fldInfo104;
...
public void setupPDM() {
  ...
  fldInfo104 = defn.addFieldInfo(104, PDMFieldType.FIX_STRING, 12, true);
  ...
}
public void receiverUsePDM(byte[] buffer) {
  ...
  String fld104Val;
  ...

  fld104Val = msg.getFieldValueAsString(fldInfo104);
}
```

Variable Field Types

The field types of STRING, UNICODE, BLOB, and MESSAGE are all variable length field types. They do not require a length to be specified when adding field info to the definition. You can use a BLOB field to store an arbitrary binary objects (in Java as an array of bytes) and a MESSAGE field to store a PDMMessage object,

which enables "nesting" PDMMessages inside other PDMMessages. Creating and using a variable length string field is nearly identical to the previous fixed string example.

```
...
private PDMFieldInfo fldInfo105;
...
public void setupPDM() {
  ...
  fldInfo105 = defn.addFieldInfo(105, PDMFieldType.STRING, true);
  ...
}

public void sourceUsePDM() {
  ...
  String fld105Val = "variable length value";
  ...
  msg.setFieldValue(fldInfo105, fld105Val);
  ...
}
```

```
...
private PDMFieldInfo fldInfo105;
...
public void setupPDM() {
  ...
  fldInfo105 = defn.addFieldInfo(105, PDMFieldType.STRING, true);
  ...
}
public void receiverUsePDM(byte[] buffer) {
  ...
  String fld105Val;
  ...

  fld105Val = msg.getFieldValueAsString(fldInfo105);
}
```

Retrieve the BLOB field values with the getFieldValueAsBlob function, and the MESSAGE field values with the getFieldValueAsMessage function.

Array Field Types

For each of the scalar field types (fixed and variable length), a corresponding array field type uses the convention ∗_ARR for the type name (ex: BOOLEAN_ARR, INT32_ARR, STRING_ARR, etc.). This lets you set and get Java values such as an int[] or string[] directly into a single field. In addition, all of the array field types can specify a fixed number of elements for the size of the array when they are defined, or if not specified, behave as variable size arrays. Do this by passing an extra parameter to the addFieldInfo function of the definition.

To be treated as a fixed-required field, an array type field must be required as well as be specified as a fixed size array of fixed length elements. For instance, a required BOOLEAN_ARR field defined with a size of 3 would be treated as a fixed-required field. Also, a required FIX_STRING_ARR field defined with a size of 5 and fixed string length of 7 would be treated as a fixed-required field. However, neither a STRING_ARR field nor a BLOB_ARR field are treated as a fixed length field even if the size of the array is specified, since each element of the array can be variable in length. In the example below, field 106 and field 108 are both treated as fixed-required fields, but field 107 is not because it is a variable size array field type.

```
...
private PDMFieldInfo fldInfo106;
private PDMFieldInfo fldInfo107;
private PDMFieldInfo fldInfo108;
...

public void setupPDM() {
  ...
  //Create information for a boolean, int32, and float field (all required)
  // as well as an optional int8 field
  ...
  //A required, fixed size array of 3 boolean elements
  fldInfo106 = defn.addFieldInfo(106, PDMFieldType.BOOLEAN_ARR, true, 3);
  //An optional, variable size array of int32 elements
  fldInfo107 = defn.addFieldInfo(107, PDMFieldType.INT32_ARR, false);
  //A required, fixed size array of 2 element which are each 5 character strings
  fldInfo108 = defn.addFieldInfo(108, PDMFieldType.FIX_STRING_ARR, 5, true, 2);
  ...
}

public void sourceUsePDM() {
  ...
  //Example values for the message
  ...
  boolean fld106Val[] = {true, false, true};
  int fld107Val[] = {1, 2, 3, 4, 5};
  String fld108Val[] = {"aaaaa", "bbbbb"};

  //Set each field value in the message
  ...
  msg.setFieldValue(fldInfo106, fld106Val);
  msg.setFieldValue(fldInfo107, fld107Val);
  msg.setFieldValue(fldInfo108, fld108Val);
  ...
}

...
private PDMFieldInfo fldInfo106;
private PDMFieldInfo fldInfo107;
private PDMFieldInfo fldInfo108;
...
public void setupPDM() {
  ...
  //Create information for a boolean, int32, and float field (all required)
  // as well as an optional int8 field
  ...
  //A required, fixed size array of 3 boolean elements
  fldInfo106 = defn.addFieldInfo(106, PDMFieldType.BOOLEAN_ARR, true, 3);
  //An optional, variable size array of int32 elements
  fldInfo107 = defn.addFieldInfo(107, PDMFieldType.INT32_ARR, false);
  //A required, fixed size array of 2 element which are each 5 character strings
  fldInfo108 = defn.addFieldInfo(108, PDMFieldType.FIX_STRING_ARR, 5, true, 2);
  ...
}

public void receiverUsePDM(byte[] buffer) {
  ...
  //Values to be retrieved from the message
  ...
  boolean fld106Val[];
  int fld107Val[];
  String fld108Val[];

  //Deserialize the bytes into a message
  msg.parse(buffer);
  //Get each field value from the message
  ...
  fld106Val = msg.getFieldValueAsBooleanArray(fldInfo106);
  if(msg.isFieldValueSet(fldInfo107)) {
    fld107Val = msg.getFieldValueAsInt32Array(fldInfo107);
  }

  fld108Val = msg.getFieldValueAsStringArray(fldInfo108);
}
```

Definition Included In Message

Optionally, a PDM message can also include the definition when it is serialized to bytes. This enables receivers to parse a PDM message without having pre-defined knowledge of the message, although including the definition with the message affects message size and performance of message deserialization. Notice that the setIncludeDefinition function is called with an argument of true for a source that serializes the definition as part of the message.

```
private PDMDefinition defn;
private PDMMessage msg;

public void setupPDM() {
  //Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);
  ...

  //Finalize the definition and create the message
  defn.finalizeDef();
  msg = new PDMMessage(defn);

  //Set the flag to indicate that the definition should also be serialized
  msg.setIncludeDefinition(true);
}
...
```

For a receiver, the setupPDM function does not need to set any flags for the message but rather should define a message without a definition, since we assume the source provides the definition. If a definition is set for a message, it will attempt to use that definition instead of the definition on the incoming message (unless the ids are different).

```
private PDMDefinition defn;
private PDMMessage msg;

public void setupPDM() {
  //Don't define a definition

  //Create a message without a definition since the incoming message will have it
  msg = new PDMMessage();
}
...
```

The PDM Field Iterator

You can use the PDM Field Iterator to check all defined message fields to see if set, or to extract their values. You can extract a field value as an Object using this method, but due to the casting involved, we recommend you use the type specific get method to extract the exact value. Notice the use of field.isValueSet to check to see if the field value is set and the type specific get methods such as getBooleanValue and getFloatValue.

```
...

public void setupPDM() {
  //Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);

  //Set the definition id and version
  defn.setId(1001);
  defn.setMsgVersMajor((byte)1);
  defn.setMsgVersMinor((byte)0);

  //Create information for a boolean, int32, and float field (all required)
  // as well as an optional int8 field
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);
  fldInfo103 = defn.addFieldInfo(103, PDMFieldType.INT8, false);
  fldInfo104 = defn.addFieldInfo(104, PDMFieldType.FIX_STRING, 12, true);
  fldInfo105 = defn.addFieldInfo(105, PDMFieldType.STRING, true);
  //A required, fixed size array of 3 boolean elements
  fldInfo106 = defn.addFieldInfo(106, PDMFieldType.BOOLEAN_ARR, true, 3);
  //An optional, variable size array of int32 elements
  fldInfo107 = defn.addFieldInfo(107, PDMFieldType.INT32_ARR, false);
  //A required, fixed size array of 2 element which are each 5 character strings
  fldInfo108 = defn.addFieldInfo(108, PDMFieldType.FIX_STRING_ARR, 5, true, 2);

  //Finalize the definition and create the message
  defn.finalizeDef();
  msg = new PDMMessage(defn);
}

public void receiveAndIterateMessage(byte[] buffer) {
  msg.parse(buffer);
  PDMFieldIterator iterator = msg.createFieldIterator();
  PDMField field = null;
  while(iterator.hasNext()) {
    field = iterator.next();
    System.out.println("Field set? " +field.isValueSet());
    switch(field.getIntName()) {
      case 100:
        boolean val100 = field.getBooleanValue();
        System.out.println("Field 100’s value is: " + val100);
        break;
      case 101:
        int val101 = field.getInt32Value();
        System.out.println("Field 101’s value is: " + val101);
        break;
      case 102:
        float val102 = field.getFloatValue();
        System.out.println("Field 102’s value is: " + val102);
        break;
      default:
        //Casting to object is possible but not recommended
        Object value = field.getValue();
        int name = field.getIntName();
        System.out.println("Field " + name + "’s value is: " + value);
        break;
    }
  }
}
```

Sample Output (106, 107, 108 are array objects as expected):

```
Field set? true
Field 100’s value is: true
Field set? true
Field 101’s value is: 7
Field set? true
Field 102’s value is: 3.14
Field set? false
Field 103’s value is: null
Field set? true
Field 104’s value is: Hello World!
Field set? true
Field 105’s value is: Variable
Field set? true
Field 106’s value is: [Z@527736bd
Field set? true
Field 107’s value is: [I@10aadc97
Field set? true
Field 108’s value is: [Ljava.lang.String;@4178460d
```

Using the Definition Cache

The PDM Definition Cache assists with storing and looking up definitions by their id and version. In some scenarios, it may not be desirable to maintain the references to the message and the definition from a setup phase by the application. A source could optionally create the definition during the setup phase and store it in the definition cache. At a later point in time, it could retrieve the definition from the cache and use it to create the message without needing to maintain any references to the objects.

```
public void createAndStoreDefinition() {
  PDMDefinition myDefn = new PDMDefinition(3, true);
  //Set the definition id and version
  myDefn.setId(2001);
  myDefn.setMsgVersMajor((byte)1);
  myDefn.setMsgVersMinor((byte)0);

  //Create information for a boolean, int32, and float field (all required)
  myDefn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  myDefn.addFieldInfo(101, PDMFieldType.INT32, true);
  myDefn.addFieldInfo(102, PDMFieldType.FLOAT, true);

  myDefn.finalizeDef();

  PDMDefinitionCache.getInstance().put(myDefn);
}

public void createMessageUsingCache() {
  PDMDefinition myFoundDefn = PDMDefinitionCache.getInstance().get(2001, 1, 0);
  if(myFoundDefn != null) {
    PDMMessage myMsg = new PDMMessage(myFoundDefn);
    //Get FieldInfo from defn and then set field values in myMsg
    //...
  }
}
```

A more advanced use of the PDM Definition Cache is by a receiver which may need to receive messages with different definitions and the definitions are not being included with the messages. The receiver can create the definitions in advance and then set a flag that allows automatic lookup into the definition cache when parsing a message (which is not on by default). Before receiving messages, the receiver should do something similar to createAndStoreDefinition (shown below) to set up definitions and put them in the definition cache. Then the flag to allow automatic lookup should be set as shown below in the call to setTryToLoadDefFromCache(true). This allows the PDMMessage to be created without a definition and still successfully parse a message by leveraging the definition cache.

```
public void createAndStoreDefinition() {
  PDMDefinition myDefn = new PDMDefinition(3, true);
  //Set the definition id and version
  myDefn.setId(2001);
  myDefn.setMsgVersMajor((byte)1);
  myDefn.setMsgVersMinor((byte)0);

  //Create information for a boolean, int32, and float field (all required)
  myDefn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  myDefn.addFieldInfo(101, PDMFieldType.INT32, true);
  myDefn.addFieldInfo(102, PDMFieldType.FLOAT, true);

  myDefn.finalizeDef();
  PDMDefinitionCache.getInstance().put(myDefn);

  //Create and store other definitions
  //...
}

public void receiveKnownMessages(byte[] buffer) {
  PDMMessage myMsg = new PDMMessage();
  //Set the flag that enables messages to try
  // looking up the definition in the cache automatically
  // when parsing a byte buffer
  myMsg.setTryToLoadDefFromCache(true);
  myMsg.parse(buffer);

  if (myMsg.getDefinition().getId() == 2001
      && myMsg.getDefinition().getMsgVersMajor() == 1
      && myMsg.getDefinition().getMsgVersMinor() == 0) {
    PDMDefinition myDefn = PDMDefinitionCache.getInstance().get(2001, 1, 0);
    PDMFieldInfo fldInfo100 = myDefn.getFieldInfo(100);
    PDMFieldInfo fldInfo101 = myDefn.getFieldInfo(101);
    PDMFieldInfo fldInfo102 = myDefn.getFieldInfo(102);

    boolean fld100Val;
    int fld101Val;
    float fld102Val;

    //Get each field value from the message
    fld100Val = myMsg.getFieldValueAsBoolean(fldInfo100);
    fld101Val = myMsg.getFieldValueAsInt32(fldInfo101);
    fld102Val = myMsg.getFieldValueAsFloat(fldInfo102);

    System.out.println(fld100Val + " " + fld101Val + " " + fld102Val);
  }
}
```

10.14.4 Migrating from SDM

Applications using SDM with a known set of message fields are good candidates for migrating from SDM to PDM. With SDM, the source typically adds fields to an SDM message without a definition. But, as shown above in the PDM examples, creating/adding a PDM definition before adding field values is fairly straightforward.

However, certain applications may be incapable of building a definition in advance due to the ad-hoc nature of their messaging needs, in which case a self-describing format like SDM may be preferred.

Simple Migration Example

The following source code shows a basic application that serializes and deserializes three fields using SDM and PDM. The setup method in both cases initializes the object instances so they can be reused by the source and receiver methods.

The goal of the sourceCreateMessageWith functions is to produce a byte array by setting field values in a message object. With SDM, actual Field classes are created, values are set, the Field classes are added to a

Fields class, and then the Fields class is added to the SDMessage. With PDM, FieldInfo objects are created during the setup phase and then used to set specific values in the PDMMessage.

The goal of the receiverParseMessageWith functions is to produce a message object by parsing the byte array and then extract the field values from the message. With SDM, the specific field is located and casted to the correct field class before getting the field value. With PDM, the appropriate getFieldValueAs function is called with the corresponding FieldInfo object created during the setup phase to extract the field value.

```
public class Migration {
  //SDM Variables
  private LBMSDMessage srcSDMMsg;
  private LBMSDMessage rcvSDMMsg;

  //PDM Variables
  private PDMDefinition defn;
  private PDMFieldInfo fldInfo100;
  private PDMFieldInfo fldInfo101;
  private PDMFieldInfo fldInfo102;
  private PDMMessage srcPDMMsg;
  private PDMMessage rcvPDMMsg;

  public static void main(String[] args) {
    Migration app = new Migration();
    System.out.println("Setting up PDM Definition and Message");
    app.setupPDM();
    System.out.println("Setting up SDM Messages");
    app.setupSDM();

    byte[] sdmBuffer;
    sdmBuffer = app.sourceCreateMessageWithSDM();
    app.receiverParseMessageWithSDM(sdmBuffer);

    byte[] pdmBuffer;
    pdmBuffer = app.sourceCreateMessageWithPDM();
    app.receiverParseMessageWithPDM(pdmBuffer);
  }

  public void setupSDM() {
    rcvSDMMsg = new LBMSDMessage();
    srcSDMMsg = new LBMSDMessage();
  }

  public void setupPDM() {
    //Create the definition with 3 fields and using int field names
    defn = new PDMDefinition(3, false);

    //Set the definition id and version
    defn.setId(1001);
    defn.setMsgVersMajor((byte)1);
    defn.setMsgVersMinor((byte)0);

    //Create information for a boolean, int32, and float field (all required)
    // as well as an optional int8 field
    fldInfo100 = defn.addFieldInfo("Field100", PDMFieldType.INT8, true);
    fldInfo101 = defn.addFieldInfo("Field101", PDMFieldType.INT16, true);
    fldInfo102 = defn.addFieldInfo("Field102", PDMFieldType.INT32, true);

    //Finalize the definition and create the message defn.finalizeDef();
    srcPDMMsg = new PDMMessage(defn);
    rcvPDMMsg = new PDMMessage(defn);
  }
  public byte[] sourceCreateMessageWithSDM() {
    byte[] buffer = null;

    LBMSDMField fld100 = new LBMSDMFieldInt8("Field100", (byte)0x42);
    LBMSDMField fld101 = new LBMSDMFieldInt16("Field101", (short)0x1ead);
    LBMSDMField fld102 = new LBMSDMFieldInt32("Field102", 12345);
    LBMSDMFields fset = new LBMSDMFields();

    try {
      fset.add(fld100);
      fset.add(fld101);
      fset.add(fld102);
    } catch (LBMSDMException e) {
      System.out.println ( e );
    }

    srcSDMMsg.set(fset);
    try {
      buffer = srcSDMMsg.data();
    } catch (IndexOutOfBoundsException e) {
      System.out.println ( "SDM Exception occurred during build of message:" );
      System.out.println ( e.toString() );
    } catch (LBMSDMException e) {
      System.out.println ( e.toString() );
    }
    return buffer;
  }

  public byte[] sourceCreateMessageWithPDM() {
    //Set each field value in the message
    srcPDMMsg.setFieldValue(fldInfo100, (byte)0x42);
    srcPDMMsg.setFieldValue(fldInfo101, (short)0x1ead);
    srcPDMMsg.setFieldValue(fldInfo102, 12345);

    //Serialize the message to bytes
    byte[] buffer = srcPDMMsg.toBytes();
    return buffer;
  }

  public void receiverParseMessageWithSDM(byte[] buffer) {
    //Values to be retrieved from the message byte fld100Val;
    short fld101Val;
    int fld102Val;

    //Deserialize the bytes into a message
    try {
      rcvSDMMsg.parse(buffer);
    } catch (LBMSDMException e) {
      System.out.println(e.toString());
    }

    LBMSDMField fld100 = rcvSDMMsg.locate("Field100");
    LBMSDMField fld101 = rcvSDMMsg.locate("Field101");
    LBMSDMField fld102 = rcvSDMMsg.locate("Field102");

    //Get each field value from the message
    fld100Val = ((LBMSDMFieldInt8)fld100).get();
    fld101Val = ((LBMSDMFieldInt16)fld101).get();
    fld102Val = ((LBMSDMFieldInt32)fld102).get();

    System.out.println("SDM Results: Field100=" + fld100Val +
                       ", Field101=" + fld101Val +
                       ", Field102=" + fld102Val);
  }

  public void receiverParseMessageWithPDM(byte[] buffer) {
    //Values to be retrieved from the message
    byte fld100Val;
    short fld101Val;
    int fld102Val;

    //Deserialize the bytes into a message
    rcvPDMMsg.parse(buffer);

    //Get each field value from the message
    fld100Val = rcvPDMMsg.getFieldValueAsInt8(fldInfo100);
    fld101Val = rcvPDMMsg.getFieldValueAsInt16(fldInfo101);
    fld102Val = rcvPDMMsg.getFieldValueAsInt32(fldInfo102);

    System.out.println("PDM Results: Field100=" + fld100Val +
                       ", Field101=" + fld101Val +
                       ", Field102=" + fld102Val);
  }
}
```

Notice that with sourceCreateMessageWithSDM function, the three fields (name and value) are created and added to the fset variable, which is then added to the SDM message. On the other hand, the sourceCreateMessage↩ WithPDM function uses the FieldInfo object references to add the field values to the message for each of the three fields.

Also notice that the receiverParseMessageWithSDM requires a cast to the specific field class (like LBMSDMField↩ Int8) once the field has been located. After the cast, calling the get method returns the expected value. On the other hand the receiverParseMessageWithPDM uses the FieldInfo object reference to directly retrieve the field value using the appropriate getFieldValueAs∗ method.

SDM Raw Classes

Several SDM classes with Raw in their name could be used as the value when creating an LBMSDMField. For example, an LBMSDMRawBlob instance could be created from a byte array and then that the LBMSDMRawBlob could be used as the value to a LBMSDMFieldBlob as shown in the following example.

```
byte[] blob = new byte[25];
LBMSDMRawBlob rawSDMBlob = new LBMSDMRawBlob(blob);
try {
  LBMSDMField fld103 = new LBMSDMFieldBlob("Field103",rawSDMBlob);
} catch (LBMSDMException e1) {
  System.out.println(e1);
}
```

The actual field named "Field103" is created in the try block using the rawSDMBlob variable which has been created to wrap the blob byte array. This field can be added to a LBMSDMFields object, which then uses it in a LBMSD↩ Message.

In PDM, there are no "Raw" classes that can be created. When setting the value for a field for a message, the appropriate variable type should be passed in as the value. For example, setting the field value for a BLOB field would mean simply passing the byte array directly in the setValue method as shown in the following code snippet since the field is defined as type BLOB.

```
private PDMFieldInfo fldInfo103;
public void setupPDM() {
  ...
  fldInfo103 = defn.addFieldInfo("Field103", PDMFieldType.BLOB, true);
  ...
  byte[] blob = new byte[25];

  srcPDMMsg.setFieldValue(fldInfo103, blob);
  ...
}
```

The PDM types of DECIMAL, TIMESTAMP, and MESSAGE expect a corresponding instance of PDMDecimal, P↩ DMTimestamp, and PDMMessage as the field value when being set in the message so those types do require an instantiation instead of using a native Java type. For example, if "Field103" had been of type PDMFieldType.DEC↩ IMAL, the following code would be used to set the value.

```
PDMDecimal decimal = new PDMDecimal((long)2, (byte)32);
srcPDMMsg.setFieldValue(fldInfo103, decimal);
```

10.15 Sending to Sources

There are many use cases where a subscriber application wants to send a message to a publisher application. For example, a client application which subscribes to market data may want to send a refresh request to the publishing feed handler. While this is possible to do with normal sources and receivers, UM supports a streamlined method of doing this.

As of UM version 6.10, a Source String can be used as a destination for sending a unicast immediate message. The UM library will establish a TCP connection to the publisher's context via its *UIM port* (also known as "request port"). The publishing application can receive this message either from a normal Receiver Object, or from a context immediate message callback via configuration options **immediate_message_topic_receiver_function (context)** or **immediate_message_receiver_function (context)** (for topicless messages).

10.15.1 Source String from Receive Event

A receiving application's receiver callback function can obtain a source's source string from the message structure. However, that string is not suitable to being passed directly to the unicast immediate message send function.

Here's a code fragment in C for receiving a message from a source, and sending a message back to the originating source. For clarity, error detection and handling code is omitted.

```c
int user_receiver_callback(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)
{
  ...
  switch (msg->type) {
    ...
  case LBM_MSG_DATA:
    /* user code which processes received message and sets up "msg_for_src" */
    ...
    /* A valid UIM destination is "SOURCE:" + source string. */
    char destination[LBM_MSG_MAX_SOURCE_LEN + 8];
    strcpy(destination, "SOURCE:");
    strcat(destination, msg->source);

    err = lbm_unicast_immediate_message(ctx, destination, NULL,   /* no topic */
                                       msg_for_src, sizeof(msg_for_src),
                                       LBM_SRC_NONBLOCK);  /* Called from context thread. */
    ...
  }  /* switch msg->type */
  ...
}  /* user_receiver_callback */
```

The **lbm_msg_t** structure supplies the source string, and **lbm_unicast_immediate_message()** is used to send a topicless immediate message to the source's context. Alternatively, a request message could be sent with **lbm_unicast_immediate_request()**. If the receive events are delivered without an event queue, then **LBM_SRC_NONBLOCK** is needed.

The example above uses the **LBM_MSG_DATA** message type. Most receiver event (message) types also contain a valid source string. Other likely candidates for this use case might be: **LBM_MSG_BOS**, **LBM_MSG_UNRECOVERABLE_LOSS**, **LBM_MSG_UNRECOVERABLE_LOSS_BURST**.

Note that in this example, a topicless message is sent. This requires the publishing application to use the **immediate_message_receiver_function (context)** option to set up a callback for receipt of topicless immediate messages. Alternatively, a topic name can be supplied to the unicast immediate message function, in which case the publishing application would either create a normal Receiver Object for that topic, or would configure a callback with **immediate_message_topic_receiver_function (context)**.

A Java program obtains the source string via **LBMMessage::source()**, and sends topicless unicast immediate messages via **LBMContext::sendTopicless()**.

A .NET implementation is essentially the same as Java.

10.15.2 Source String from Source Notification Function

Some subscribing applications need to send a message to the publisher as soon as possible after the publisher is subscribed. Receiver events can sometimes take significant time to be delivered. The source string can be obtained via the **source_notification_function (receiver)** configuration option. This defines a callback function which is called at the start of the process of subscribing to a source.

Here's a code fragment in C for sending a message to a newly-discovered source. For clarity, error detection and handling code is omitted.

During initialization, when the receiver is defined, the callback must be configured using the **lbm_rcv_src_notification_func_t_stct** structure:

```
lbm_rcv_src_notification_func_t src_notif_callback_info;
src_notif_callback_info.create_func = src_notif_callback_create;  /* User function. */
src_notif_callback_info.delete_func = src_notif_callback_delete;  /* User function. */
src_notif_callback_info.clientd = NULL;   /* Can be user's receiver-specific state. */
...
lbm_rcv_topic_attr_t *rcv_topic_attr;
err = lbm_rcv_topic_attr_create_from_xml(&rcv_topic_attr, "MyCtx", receiver_topic_name);

err = lbm_rcv_topic_attr_setopt(rcv_topic_attr, "source_notification_function",
                               &src_notif_callback_info, sizeof(src_notif_callback_info));

lbm_topic_t *receiver_topic;
err = lbm_rcv_topic_lookup(&receiver_topic, ctx, receiver_topic_name, rcv_topic_attr);

lbm_rcv_t *receiver;
err = lbm_rcv_create(&receiver, ctx, receiver_topic, ...);
```

This creates the Receiver Object with the source notification callback configured. Note that the source notification callback has both a create and a delete function, to facilitate state management by the user.

```
void * src_notif_callback_create(const char *source_name, void *clientd)
{
  /* This function is called when the subscription is being set up. */

  /* user code which sets up "msg_for_src" */
  ...
  /* A valid UIM destination is "SOURCE:" + source string. */
  char destination[LBM_MSG_MAX_SOURCE_LEN + 8];
  strcpy(destination, "SOURCE:");
  strcat(destination, source_name);

  err = lbm_unicast_immediate_message(ctx, destination, NULL,   /* no topic */
                                    msg_for_src, sizeof(msg_for_src),
                                    LBM_SRC_NONBLOCK);  /* Called from context thread. */
  ...
  return NULL;  /* Can be per-source state. */
}  /* src_notif_callback_create */


int src_notif_callback_delete(const char *source_name, void *clientd, void *source_clientd) {
  /* This function not used for anything in this example, but could be used to
   * to clean up per-source state. */
  return 0;
}  /* src_notif_callback_delete */
```

A Java program configures the source notification callback via **com::latencybusters::lbm::LBMReceiverAttributes::setSourceNotificationCallbacks**.

A .NET implementation is essentially the same as Java.

10.15.3 Sending to Source Readiness

In most use cases for sending messages to a source, there is an implicit assumption that a subscribing receiver is fully set up and ready to receive messages from the publisher. However, due to the asynchronous nature of UM, there is no straight-forward way for a receiver to know the earliest point in time when messages sent by the source will be delivered to the receiver. For example, in a routed network (using the DRO), a receiver might deliver BOS to the application, but that just means that the connection to the proper DRO is complete. There could still be delays in the entire end-to-end path being able to deliver messages.

Also, be aware that although unicast immediate messages are delivered via TCP, these messages are not guaranteed. Especially in a routed network, there exists the possibility that a message will fail to reach the publisher.

In most cases, the immediate message is received by the publisher, and by the time the publisher reacts, the end-to-end source-to-receiver path is active. However, in the unlikely event that something goes wrong, a subscribing application should implement a timeout/retry mechanism. This advice is not specific to the "sending to source" use cases, and should be built into any kind of request/response-oriented use case.

10.16 Spectrum

UM Spectrum, which refers to a "spectrum of channels", allows the application designer to sub-divide a topic into any number of channels, which can be individually subscribed to by a receiving application. This provides an extra level of message filtering.

The publisher first allocates the desired number of source channel objects using **lbm_src_channel_create()**. Then it creates a topic source in the normal way. Finally, the application sends messages using **lbm_src_send_ex()**, specifying the source channel object in the **lbm_src_send_ex_info_t**'s channel_info field.

A receiving application first creates a topic receiver in the normal way. Then it subscribes to channels using **lbm_rcv_subscribe_channel()** or lbm_wrcv_subscribe_channel(). Since each channel requires a different receiver callback, the receiver application can achieve more granular filtering of messages. Moreover, messages are received in-order across channels since all messages are part of the same topic stream.

It should be noted that a regular topic receiver (one for which no spectrum channels are subscribed) delivers all received messages from a matching spectrum topic source to the receiver's callback without creating the channel_info object.

You can accomplish the same level of filtering with a topic space design that creates separate topics for each channel, however, UM cannot guarantee the delivery of messages from multiple sources/topics in any particular order. Not only can UM Spectrum deliver the messages over many channels in the order they were sent by the source, but it also reduces topic resolution traffic since UM advertises only topics, not channels.

Note

> With the UMQ product, you cannot use UM Spectrum with Queuing (both Brokered and ULB).

10.16.1 Spectrum Performance Advantages

The use of separate callbacks for different channels improves filtering and also relieves the source application of the task of including filtering information in the message data.

Java and .NET performance also receives a boost because messages not of interest can be discarded before they transition to the Java or .NET level.

10.16.2 Spectrum Configuration Options

Spectrum's default behavior delivers messages on any channels the receiver has subscribed to on the callbacks specified when subscribing, and all other messages on the receiver's default callback. This behavior can be changed with the following configuration options.

- **null_channel_behavior (receiver)** - behavior for messages delivered with no channel information.
- **unrecognized_channel_behavior (receiver)** - behavior for messages delivered with channel information but are on a channel for which the receiver has not registered interest.
- **channel_map_tablesz (receiver)** - controls the size of the table used by a receiver to store channel subscriptions.

10.16.3 Spectrum Receiver Callback

When an application subscribes to a spectrum channel, it uses the spectrum subscribe API:

- **lbm_rcv_subscribe_channel()** - C/C++
- **com::latencybusters::lbm::LBMReceiver::subscribeChannel()** - Java/.NET

Note that when subscribing to a channel, the receiver callback function is optional. If null is supplied as the callback, UM will invoke the underlying receiver's callback.

If a separate callback is supplied for the channel, be aware that only data message event types (**LBM_MSG_DATA**, **LBM_MSG_REQUEST**) will be delivered to it. Non-data events (**LBM_MSG_BOS**, **LBM_MSG_EOS**, **LBM_MSG_UNRECOVERABLE_LOSS**, etc.) will be delivered to the underlying receiver's callback.

10.16.4 Smart Sources and Spectrum

Smart Sources support Spectrum, but via different API functions. You need to tell UM that you intend to use spectrum at Smart Source creation time using the **smart_src_enable_spectrum_channel (source)** configuration option. This pre-allocates space in the message header for the spectrum channel.

With Smart Sources, there is no need to allocate a Spectrum source object with **lbm_src_channel_create()**. Instead, you simply set the **LBM_SSRC_SEND_EX_FLAG_CHANNEL** flag and the spectrum channel number in the **lbm_ssrc_send_ex_info_t** passed to the **lbm_ssrc_send_ex()** API function. It is also usually necessary to tell UM to rebuild the header. For example:

```
lbm_ssrc_send_ex_info_t ss_send_info;
memset((char *)&ss_send_info, 0, sizeof(ss_send_info));
ss_send_info.flags = LBM_SSRC_SEND_EX_FLAG_CHANNEL |
      LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
ss_send_info.channel = desired_channel_number;

err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
```

Note that if you are sending multiple messages in a row to the same spectrum channel, you can get a small performance boost by leaving the header alone and code subsequent calls as:

```
lbm_ssrc_send_ex_info_t ss_send_info;
ss_send_info.flags = LBM_SSRC_SEND_EX_FLAG_CHANNEL;

err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
```

When a Smart Source is created with Spectrum enabled, it is possible to send messages without a Spectrum channel, either by clearing the LBM_SSRC_SEND_EX_FLAG_CHANNEL flag in **lbm_ssrc_send_ex_info_t**, or by simply not supplying a **lbm_ssrc_send_ex_info_t** object by passing NULL for the `info` parameter. This suppresses all features enabled by that structure.

Note

> If using both Spectrum and Message Properties with a single Smart Source, there is an added restriction: it is not possible to send a message omitting only one of those features. I.e. if both are enabled when the Smart Source is created, it is not possible to send a message with a message property and not a channel, and it is not possible to send a message with a channel and not a property. This is because the message header is defined at Smart Source creation, and the header either must contain both or neither.

10.17 Hot Failover (HF)

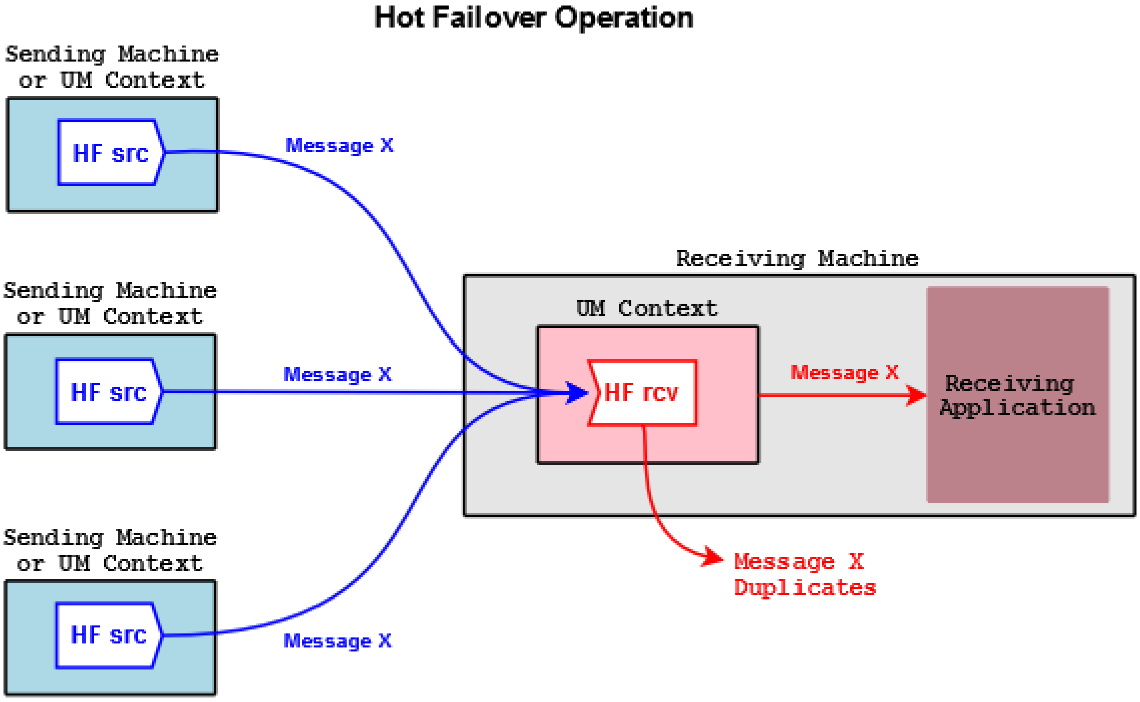
UM Hot Failover (HF) lets you implement sender redundancy in your applications. You can create multiple HF senders in different UM contexts, or, for even greater resiliency, on separate machines. There is no hard limit to the number of HF sources, and different HF sources can use different transport types.

Note

> With the UMQ product, you cannot use Hot Failover with Queuing (both Brokered and ULB).

Hot Failover receivers filter out the duplicate messages and deliver one message to your application. Thus, sources can drop a few messages or even fail completely without causing message loss, as long as the HF receiver receives each message from at least one source.

The following diagram displays Hot Failover operation.



In the figure above, HF sources send copies of Message X. An HF receiver delivers the first copy of Message X it receives to the application, and discards subsequent copies coming from the other sources.

10.17.1 Implementing Hot Failover Sources

You create Hot Failover sources with **lbm_hf_src_create()**. This returns a source object with internal state information that lets it send HF messages. You delete HF sources with the **lbm_src_delete()** function.

HF sources send HF messages via **lbm_hf_src_send_ex()** or **lbm_hf_src_sendv_ex()**. These functions take a sequence number, supplied via the exinfo object, that HF receivers use to identify the same message sent from different HF sources. The exinfo has an hf_sequence_number, with a flag (LBM_SRC_SEND_EX_FLAG_HF_32 or LBM_SRC_SEND_EX_FLAG_HF_64) that identifies whether it's a 32- or 64-bit number. Each HF source sends the same message content for a given sequence number, which must be coordinated by your application.

If the source needs to restart its sequence number to an earlier value (e.g. start of day; not needed for normal wraparound), delete and re-create the source and receiver objects. Without re-creating the objects, the receiver sees the smaller sequence number, assumes the data are duplicate, and discards it. In (and only in) cases where this cannot be done, use **lbm_hf_src_send_rcv_reset()**.

Note

> Your application must synchronize calling **lbm_hf_src_send_ex()** or **lbm_hf_src_sendv_ex()** with all threads sending on the same source. (One symptom of not doing so is messages appearing at the receiver as inside intentional gaps and being erroneously discarded.)

Please be aware that non-HF receivers created for an HF topic receive multiple copies of each message. We recommend you establish local conventions regarding the use of HF sources, such as including "HF" in the topic name.

For an example source application, see **Example lbmhfsrc.c**.

10.17.2 Implementing Hot Failover Receivers

You create HF receivers with **lbm_hf_rcv_create()**, and delete them using **lbm_hf_rcv_delete()** and **lbm_hf_rcv_delete_ex()**.

Incoming messages have an hf_sequence_number field containing the sequence number, and a message flag (LBM_MSG_FLAG_HF_32 or LBM_MSG_FLAG_HF_64) noting the bit size.

Note

> Previous UM versions used sequence_number for HF message identification. This field holds a 32-bit value and is still set for backwards compatibility, but if the HF sequence numbers are 64-bit lengths, this non-HF sequence number is set to 0. Also, you can retrieve the original (non-HF) topic sequence number via lbm_msg_retrieve_original_sequence_number() or, in Java and .NET, via LBMMessage.osqn().

For the maximum time period to recover lost messages, the HF receiver uses the minimum of the LBT-RM and LBT-RU NAK generation intervals (**transport_lbtrm_nak_generation_interval (receiver)**, **transport_lbtru_nak_generation_interval (receiver)**). Each transport protocol is configured as normal, but the lost message recovery timer is the minimum of the two settings.

Some lbm_msg_t objects coming from HF receivers may be flagged as having "passed through" the HF receiver. This means that the message has not been ordered with other HF messages. These messages have the LBM_MSG_FLAG_HF_PASS_THROUGH flag set. UM flags messages sent from HF sources using **lbm_src_send()** in this manner, as do all non-HF sources. Also, UM flags EOS, no source notification, and requests in this manner as well.

For an example receiver application, see **Example lbmhfrcv.c**.

10.17.3 Implementing Hot Failover Wildcard Receivers

To create an HF wildcard receiver, set option **hf_receiver (wildcard_receiver)** to 1, then create a wildcard receiver with **lbm_wildcard_rcv_create()**. This actually creates individual HF receivers on a per-topic basis, so that each topic can have its own set of HF sequence numbers. Once the HF wildcard receiver detects that all sources for a particular topic are gone it closes the individual topic HF receivers and discards the HF sequence information (unlike a standard HF receiver). You can extend or control the delete timeout period of individual HF receivers with option **resolver_no_source_linger_timeout (wildcard_receiver)**.

10.17.4 Java and .NET

For information on implement the HF feature in a Java application, go to UM Java API and see the documentation for classes **LBMHotFailoverReceiver** and **LBMHotFailoverSource**.

For information on implement the HF feature in a .NET application, go to UM .NET API and navigate to Namespaces->**com.latencybusters.lbm**->LBMHotFailoverReceiver and LBMHotFailoverSource.

10.17.5 Using Hot Failover with Persistence

When implementing Hot Failover with Persistence, you must consider the following impact on hardware resources:

- Additional storage space required for a persistent Store
- Higher disk activity
- Higher network activity
- Increased application complexity regarding message filtering

Also note that you must enable **Explicit Acknowledgments** and "Hot Failover duplicate delivery" (**hf_duplicate_delivery (receiver)**) in each Hot Failover receiving application.

For detailed information on using Hot Failover with Persistence, see the Knowledge Base article `FAQ: Is UMP compatible with Hot Failover?`

10.17.6 Hot Failover Intentional Gap Support

UM supports intentional gaps in HF message streams. Your HF sources can supply message sequence numbers with number gaps up to 1073741824. HF receivers automatically detect the gaps and consider any missing message sequence numbers as not sent and do not attempt recovery for these missing sequence numbers. See the following example.

1. HF source 1 sends message sequence numbers: 10, 11, 12, 13, 25, 26, 38
2. HF source 2 sends message sequence numbers: 10, 11, 12, 13, 25, 26, 38

HF receiver 1 receives message sequence numbers in order with no pause between any messages: 10, 11, 12, 13, 25, 26, 38

10.17.7 Hot Failover Optional Messages

Hot Failover sources can send optional messages that HF receivers can be configured to receive or not receive (**hf_optional_messages (receiver)**). HF receivers detect an optional message by checking **lbm_msg_t.flags** for LBM_MSG_FLAG_HF_OPTIONAL. HF sources indicate an optional message by passing **LBM_SRC_SEND_EX_FLAG_HF_OPTIONAL** in the **lbm_src_send_ex_info_t.flags** field to **lbm_hf_src_send_ex()** or **lbm_hf_src_sendv_ex()**. In the examples below, optional messages appear with an "o" after the sequence number.

1. HF source 1 sends message sequence numbers: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o, 19o, 20

2. HF source 2 sends message sequence numbers: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o, 19o, 20

HF receiver 1 receives: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o, 19o, 20

HF receiver 2, configured to ignore optional messages, receives: 10, 11, 12, 15, 20

10.17.8 Using Hot Failover with Ordered Delivery

An HF receiver takes some of its operating parameters directly from the receive topic attributes. The **ordered_delivery (receiver)** setting indicates the ordering for the HF receiver.

Note

> UM supports Arrival Order with HF only when all sources use the same transport type.

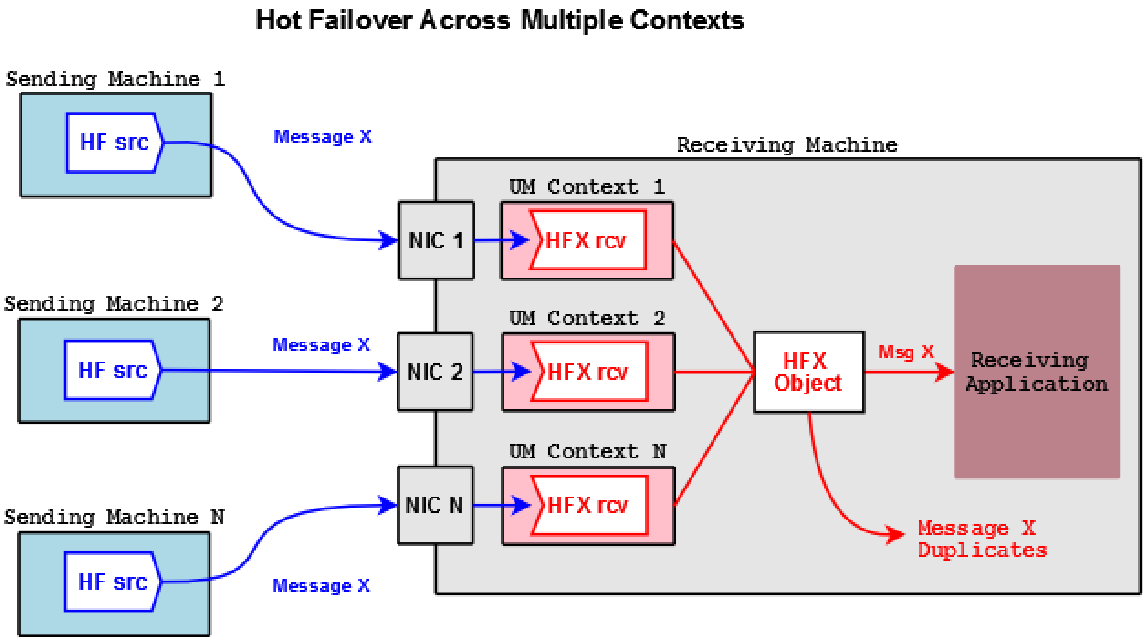
10.17.9 Hot Failover Across Multiple Contexts (HFX)

Warning

> **The HFX feature is deprecated and may be removed in a future release**. Users of HFX are advised to `contact Informatica Support`.

If you have a receiving application on a multi-homed machine receiving HF messages from HF sources, you can set up the Hot Failover Across Contexts (HFX) feature. This involves setting up a separate UM context to receive HF messages over each NIC and then creating an HFX Object, which drops duplicate HF messages arriving over all contexts. Your receiving application then receives only one copy of each HF message. The HFX feature achieves the same effect across multiple contexts as the normal Hot Failover feature does within a single context.

The following diagram displays Hot Failover operation across UM contexts.



For each context that receives HF messages, create one HFX Receiver per topic. Each HFX Receiver can be configured independently by passing in a UM Receiver attributes object during creation. A unique client data pointer can also be associated with each HFX Receiver. The HFX Object is a special Ultra Messaging object and does not live in any UM context.

Note: You never have to call lbm_topic_lookup() for a HFX Receiver. If you are creating HFX Receivers along with normal UM receivers for the same topic, do not interleave the calls. For example, call **lbm_hfx_create()** and **lbm_hfx_rcv_create()** for the topic. Then call lbm_topic_lookup() and **lbm_rcv_create()** for the topic to create the normal UM receivers.

The following outlines the general procedure for HFX.

1. Create an HFX Object for every HF topic of interest with **lbm_hfx_create()**, passing in an attributes object created with **lbm_hfx_attr_create_from_xml()** to specify any attributes desired.

2. Create a context for the first NIC receiving HF messages with **lbm_context_create()**.

3. Create a HFX Receiver for every HF topic with **lbm_hfx_rcv_create()**, passing in UM Receive Topic Attributes.

4. Repeat steps 2 and 3 for all NICs receiving HF message

5. Receive messages. The HFX Object identifies and drops all duplicates, delivering messages through a single callback (and optional event queue) specified when you created the HFX Object.

Delete each HFX Receiver with **lbm_hfx_rcv_delete()** or **lbm_hfx_rcv_delete_ex()**. Delete the HFX Object with **lbm_hfx_delete()**.

Note

> When writing source-side HF applications for HFX, be aware that HFX receivers do not support hf_sequence, 64-bit sequence numbers, the **lbm_hf_src_send_rcv_reset()** function, or HF wildcard receivers. See **Hot Failover Operation Options**, especially HFX-specific options.

10.18 NAK Cutoff

The "NAK cutoff" feature allows an application to force its context(s) to stop sending NAKs (LBT-RM only), even if packet loss is detected.

This feature was first added to UM in version 6.16.1 with the C API **lbm_context_set_nak_cutoff**.

10.18.1 Why NAK Cutoff?

NAK-based network protocols, like LBT-RM and LBT-RU, can be susceptible to NAK storms. UM's algorithms enable the user to control the risk of NAK storms, reducing it close to zero if desired. However, be aware that these algorithms present a trade-off between performance and stability that needs to be considered.

Many users prevent NAK storms by simply provisioning their systems to be able to handle their worst-case load. In this case, there can be no overload (unless the calculation of "worst case" was faulty), and without overload, there can be no NAK storm.

But for some users, designing for theoretical worst-case is not practical, so other strategies must be used. For example, to reduce the chances of a NAK storm close to zero, it might be necessary to establish Rate Controls that will, under periods of stress, limit a publisher's ability to send messages when it is ready to. This is essentially an intentional use of latency to prevent overload.

But again, many of our customers consider those latency costs too high, so they live with increased risk. For example, they may establish rate controls that "overbook" the network's capacity under the assumption that not *all* publishers will send at their maximum allowed rate at the same time. This is generally a safe assumption, but still leaves open the possibility of overload.

UM version 6.16.1 introduced a feature called the NAK Cutoff. This is an API that allows an application to force its context(s) to stop sending NAKs, even if packet loss is detected. This API may be useful to users who cannot reduce the changes of NAK storms low enough, and want a contingency for the unlikely event that a NAK storm does happen. Note that employing this feature will result in some amount of Unrecoverable Loss.

If the operations team concludes that they are having a NAK storm, they can command their applications to invoke the NAK cutoff feature (call the API) to disable NAKS long enough to allow the network to stabilize.

Attention

> This API does not control remote contexts. Only the context(s) of the application calling it are affected. In most UM-based systems, there is an independent network used for command and control, separate from the low-latency network used for application data. It is the user's responsibility to implement centralized command and control to instruct their applications to call the cutoff API.

10.18.2 What Is a NAK Storm?

When applications send NAKs, that is typically not a NAK storm. NAKs are simply part of the UM lost packet recovery protocols. But in a worst-case scenario, the increased load of the lost-packet recovery protocols can cause additional packet loss, leading to a self-reinforcing feedback loop where loss triggers loss recovery, which causes additional loss, which triggers more loss recovery, etc.

This self-reinforcing feedback loop is a NAK storm. If not controlled, they can cripple a network for minutes or even hours, potentially requiring a shutdown / restart of the core parts of your system.

See `NAK Storms` for detailed explanation of NAK storm causes, prevention, and avoidance.

10.18.3 NAK Storm Prevention

You can reduce the chances of a NAK storm by avoiding packet loss. Since the vast majority of packet loss is caused by overload, the basic methods of avoiding packet loss are to maximize the efficiency of data consumers and to control the sending of messages by data sources.

See `Packet Loss` for detail and loss avoidance strategies.

See `Configuring UDP-Based Transports` for specific UM configuration advice.

10.19 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**.

The Persistence Store daemon and the DRO daemon each have a simple web server which provides operational information. This information is important for monitoring the operation and performance of these daemons. However, while the web-based presentation is convenient for manual, on-demand monitoring, it is not suitable for automated collection and recording of operational information for historical analysis.

(The UMDS product also supports Daemon Statistics as of UMDS version 6.12; see UMDS documentation for details.)

Starting with UM version 6.11, a feature called "Daemon Statistics" has been added to the Store and DRO daemons. The Stateful Resolver Service (SRS), added in UM version 6.12, supports Daemon Statistics only (no web server). The Daemon Statistics feature supports the background publishing of their operational information via UM messages. Monitoring systems can now subscribe to this information in much the same way that UM transport statistics can be subscribed.

While the information published by the Store, DRO, and SRS daemon statistics differ in their content, the general feature usage is the same between them. When the feature is configured, the daemon will periodically collect and publish its operational information.

The following sections give general information which is common across daemons, followed by links to daemon-specific details.

10.19.1 Daemon Controller

With the introduction of the Daemon Statistics feature, a new context is added to the daemons: the Daemon Controller. This context publishes the statistics and also can be configured to accept daemon control requests from external applications. These control requests are primarily used for controlling the Daemon Statistics feature (see Daemon Control Requests), but are also used for a few daemon-specific control functions that are unrelated to Daemon Statistics (for example, **Request: Mark Stored Message Invalid**).

Note that in every UM component that supports the Daemon Statistics feature, the Daemon Controller defaults to disabled. Each component's Daemon Statistics configuration must be set to enable the function of the Daemon Controller.

10.19.2 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 operational information is published as messages of different types sent over a normal UM topic source (topic name configurable). For the Store and DRO daemons, each message is in the form of a binary, C-style data structure.

There are generally two categories of messages: *config* and *stats*. A given instance of a category "config" message does not have content which changes over time. An instance of a category "stats" message has content that *does* change over time. The daemon-specific documentation indicates which messages are in which category.

Each message type is configured for a publishing interval. When the publishing interval for a message type expires, the possible messages are checked to see if its content has materially changed since the last interval. If not, then the message is *not* republished. The publishing interval for a stat message is typically set to shorter periods to see those changes as they occur.

Note that the SRS message format is JSON, and therefore the granularity of published data is finer. I.e. a given message type might be published, but only a subset of the fields within the message might be included. In contrast, the daemons which publish binary structures send the structures complete.

Finally, note that while the contents of a given instance of a config message does not change over time, new instances of the message type can be sent as a result of state changes. For example, a new instance of **umestore↩ _repo_dmon_config_msg_t** is published each time a new source registers with the Store.

More detailed information is available in the daemon-specific documentation referenced below.

10.19.3 Daemon Statistics Binary Data

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**.

For the Store and DRO daemons, the messages published are in binary form and map onto the C data structures defined for each message type.

For the SRS service, the messages are formatted as JSON, so this section does not apply to the SRS.

The byte order of the structure fields is defined as the host endian architecture of the publishing daemon. Thus, if a monitoring host receiving the messages has the same endian architecture, the binary structures can be used directly. If the monitoring host has the opposite endian architecture, the receiver must byte-swap the fields.

The message structure is designed to make it possible for a monitoring application to detect a mismatch in endian architecture. Detection and byte swapping is demonstrated with daemon-specific example monitoring applications.

More detailed information is available in the daemon-specific documentation referenced below.

10.19.4 Daemon Statistics Versioning

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**.

For the Store and DRO daemons, each message sent by the daemon consists of a standard header followed by a message-type-specific set of fields. The standard header contains a `version` field which identifies the version of the C include file used to build the daemon.

For the SRS service, the messages are formatted as JSON, so this section does not apply to the SRS.

For example, the Store daemon is built with the include file **umedmonmsgs.h**. With each daemon statistics message sent by the Store daemon, it sets the header version field to **LBM_UMESTORE_DMON_VERSION**. With each new release of the UM package, if that include file changes in a substantive way, the value of **LBM_UMESTORE_↩ DMON_VERSION** is increased. In this way, a monitoring application can determine if it is receiving messages from a Store daemon whose data structures match the monitoring application's structure definitions.

More detailed information is available in the daemon-specific documentation referenced below.

10.19.5 Daemon Control Requests

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**.

Each daemon publishing binary daemon stats can optionally be configured to accept command-and-control requests from administrative applications. These command-and-control requests are handled by the daemon's Daemon Controller, a context added to support Daemon Statistics.

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, specially if "config" requests are enabled, Informatica recommends Securing Daemon Control Requests.

There are different categories of these requests. All daemons have in common categories *"snapshot"* and *"config"*, which are related to Daemon Statistics. Other categories are specific to the daemon type.

"Snapshot" requests tell the daemon to immediately republish the desired stats and/or configs without waiting until the next publishing interval. These requests might be sent by a monitoring application which has only just started running and needs a full snapshot of the operational information.

"Config" requests tell the daemon to modify an operational parameter of the running daemon.

An application sends a request to the daemon, and the daemon sends status messages in response. The exchanges are made via standard UM topicless immediate Request/Response messaging. These requests should be sent using the Unicast Immediate Messaging (UIM) API for sending the requests using **lbm_unicast_immediate_request()**. See Unicast Immediate Messaging for details on UIM.

To use UIM effectively, Informatica recommends configuring the daemon monitor context for a specific UIM interface and port using: **request_tcp_port (context)** and **request_tcp_interface (context)**. This enables the monitoring application to know how to address the request UIMs to the proper daemon.

For the Store and DRO daemons, The request message is formatted as a simple ASCII string. For the SRS service, the request message is formatted as a JSON message. The request is sent as a topicless unicast immediate request message. The daemon reacts by parsing the request and sending a UM response with a success/failure response. If the request was parsed successfully, the daemon then performs the requested operation (republishing the data or modifying the operational parameter). There are daemon-specific example applications which demonstrate the use of this request feature.

More detailed information is available in the daemon-specific documentation referenced below.

10.19.6 Securing Daemon Control Requests

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**.

UM Daemon Statistics are implemented using normal UM messaging. In particular, the Daemon Controller capability allows a remote application to send requests to a daemon that modify its behavior. If misused, these behaviors can be disruptive to normal operation. For example, the "umedmon" program can instruct a persistent Store to flag stored messages as invalid, which would prevent their delivery to a recovering receiver.

Note that the UM daemons default to rejecting these command-and-control messages, so taking special security precautions is only necessary if you have configured daemons to enable the Daemon Control requests.

One common way to prevent unauthorized use is to tightly control access to your production network so that no unauthorized users can accidentally or maliciously use Daemon Control requests to interfere with normal operation.

Additionally, you can configure the Daemon Control context for encryption, which supports certificate-based access control. This requires the use of an encrypted TRD for Daemon Statistics. If your normal message data is unencrypted, you will need to define one or more *new* TRDs for Daemon Statistics that are separate from the normal TRDs (an encrypted TRD must not have unencrypted contexts assigned to it).

Use the **use_tls (context)** configuration option in the UM configuration file you supply for the Daemon Statistics. For example, for the Store Daemon use the **UMP Element "<lbm-config>"** contained within the **UMP Element "<daemon-monitor>"**.

Note that the use of an encrypted TRD will require that the Daemon Statistics data be configured for the TCP transport.

Since UM's encryption feature is certificate-based. A user wanting to use a Daemon Control request tool must have access to the proper certificate file(s). This means that unauthorized users must *not* have access to the certificate file(s).

Finally, be aware that a partially-encrypted DRO network can break the security of an encrypted TRD; see TLS and the DRO.

See Encrypted TCP for details on using encryption.

10.19.7 Daemon Statistics Details

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**.

For details on the persistent Store's daemon statistics feature, see **Store Binary Daemon Statistics**.

For details on the DRO's daemon statistics feature, see **DRO Binary Daemon Statistics**.

For details on the SRS's daemon statistics feature, see SRS Daemon Statistics.

Chapter 11

Advanced Optimizations

The internal design of UM has many compromises between performance and flexibility. For example, there are critical sections which maintain state information which must be kept internally consistent. Since UM allows the application the flexibility of multi-threaded use, those critical sections are protected with Mutex locks. These locks add very little overhead to UM's execution, but "very little" is not the same as "zero". The use of locks is a compromise between efficiency and flexibility. Similar lines of reasoning explain why UM makes use of dynamic memory (malloc and free), and bus-interlocked read/modify/write operations (e.g. atomic increment).

UM provides configuration options which improve efficiency, at the cost of reduced application design flexibility. Application designers who are able to constrain their programs within certain restrictions can take advantage of improved performance and reduced latency outliers (jitter).

RECEIVE-SIDE

- Receive Thread Busy Waiting.
- Receive Buffer Recycling.
- Single Receiving Thread - not generally applicable.
- Extended Context Process Events - not generally applicable.
- Receive Multiple Datagrams.
- Transport Demultiplexer Table Size.
- XSP Latency Reduction.
- Receive-Side Batching - for use with event queues.

SEND-SIDE

- Smart Sources.
- Zero-Copy Send API (not recommended; see Comparison of Zero Copy and Smart Sources).

GENERAL (both receivers and senders)

- Core Pinning.
- Memory Latency Reduction.
- Zero Object Delivery (Java and .NET).

11.1 Receive Thread Busy Waiting

Busy looping is a method for reducing latency and especially latency outliers (jitter) by preventing threads from going to sleep. In an event-driven system, if a thread goes to sleep waiting for an event, and the event happens, the operating system needs to wake the thread back up and schedule its execution on a CPU core. This can take several microseconds. Alternatively, if the thread avoids going to sleep and "polls" for the event in a tight loop, it will detect and be able to start processing the event without the operating system scheduling delay.

However, remember that a thread that never goes to sleep will fully consume a CPU core. If you have more busy threads than you have CPU cores in your computer, you can have CPU thrashing, where threads are forced to time-share the cores. This can produce *worse* latency than sleep-based waits.

Only use busy waiting if there are enough cores to allocate a core exclusively to each busy thread. Also, pinning threads to cores is highly recommended to prevent thread migration across cores, which can introduce latency and significant jitter.

11.1.1 Network Socket Busy Waiting

The UM receive code performs socket calls to wait for network events, like received data. By default, UM does sleep-based waiting for events. For example, if there are no packets waiting to be read from any of the sockets, the operating system will put the receive thread to sleep until a packet is received.

However, the **file_descriptor_management_behavior (context)** configuration option can be used to change the behavior of the receive thread to *not* sleep. Instead, the socket is checked repeatedly in a tight loop - busy waiting. With most use cases, enabling "busy wait" will typically reduce *average* latency only a little, but it can significantly reduce latency outliers (jitter).

For network-based transports, a receive thread can either be the main context thread, or it can be an XSP thread. A given application can have more than one context, and a given context can have zero or more XSPs. The threads of each context and XSP can be independently configured to have either busy waiting or sleep waiting.

Note that when creating an XSP, it is not unusual to simply let the XSP inherit the parent context's attributes. However, a common XSP use case is to create a single XSP for user data, and leave the parent context for Topic Resolution and other overhead. In this case, you may want to configure the parent context to use sleep-based waiting ("pend"), and configure the XSP to use busy waiting ("busy-wait"). You will need to pass a context attribute to the **lbm_xsp_create()** API.

A Better Alternative

Kernel bypass network drivers typically have a busy waiting mode of operation which happens inside the driver itself. For example Solarflare's Onload driver can be configured to do busy waiting. This can produce even greater improvement than UM receive thread busy waiting. When using a busy waiting kernel bypass network driver like Onload, the **file_descriptor_management_behavior (context)** configuration option should be left at its default, "pend".

11.1.2 IPC Transport Busy Waiting

The Transport LBT-IPC does not use the context or XSP threads for receiving messages. It has its own internal thread which can be configured for busy waiting with the **transport_lbtipc_receiver_thread_behavior (context)** option.

11.1.3 SMX Transport Busy Waiting

The Transport LBT-SMX does not use the context or XSP threads for receiving messages. It has its own internal thread which always operates in busy waiting. I.e. it cannot be configured to sleep waiting.

11.2 Zero Object Delivery

Zero Object Delivery is a set of Java and .NET programming conventions to avoid garbage.

In Java and .NET, garbage collection (GC) can be a very useful feature to simplify programming. However, for low latency applications, GC is a problem. Most JVMs impose milliseconds of latency when GC runs. Also, unnecessary object creation introduces additional overhead.

Most high-performance Java and .NET programs avoid GC by saving objects that are no longer needed and reusing them. Most of this is the responsibility of the application programmer. However, UM application callbacks represent a special case. When UM needs to deliver an event, like a received message, to a Java or .NET program, it typically passes to the callback one or more objects containing the pertinent event data. UM needs to know when the application is finished with those passed-in objects so that UM can re-use them.

This requires the application to follow a set of conventions. Applications that deviate from these conventions will typically suffer from higher latency due to unnecessary object creation and garbage collection. These programming conventions are collectively called "Zero Object Delivery" (ZOD).

The ZOD programming convention are the same between Java and .NET. (ZOD does not apply to the C API.)

1. Special data access methods must be used to access the received message's data.
2. If a message needs to be processed outside of the application receiver callback, it must be "promoted". A UM recycler must be used to ensure no garbage is created.
3. Every message must be explicitly deleted when the application is finished processing it. This is true for messages fully processed by the receiver callback, and for promoted messages that are processed outside of the receiver callback. This is done by calling the "dispose()" method.

Details of these coding conventions can be found at:

- Java Message Reception
- .NET Message Reception

Note

> While the coding conventions of ZOD are most often seen with the receiver callback, many of the same conventions exist with other event delivery callbacks. For example, source events should be disposed. Source events must also be promoted before they are passed to another thread for processing, after which they should be disposed and recycled.

11.3 Receive Buffer Recycling

By default, the UM receive code base allocates a fresh buffer for each received datagram. This allows the user a great degree of threading and buffer utilization flexibility in the application design.

For transport types RM (reliable multicast), RU (Reliable Unicast), and IPC (shared memory), you can set a configuration option to enable reuse of receive buffers, which can avoid per-message dynamic memory calls (malloc/free). This produces a modest reduction in average latency, but more importantly, can significantly reduce occasional latency outliers (jitter).

See the configuration options:

- RM - **transport_lbtrm_recycle_receive_buffers (context)**
- RU - **transport_lbtru_recycle_receive_buffers (context)**
- IPC - **transport_lbtipc_recycle_receive_buffers (context)**

Note that setting the option does not guarantee the elimination of per-message malloc and free except in fairly restrictive use cases.

Also note that this feature is different from the Java and .NET ZOD recycler. See Zero Object Delivery.

11.3.1 Receive Buffer Recycling Restrictions

There are no hard restrictions to enabling buffer recycling. I.e. it is not functionally not compatible with any use patterns or UM features. However, some use patterns will prevent the recycling of the receive buffer, and therefore not deliver the benefit, even if the configuration option is set.

- **Event Queues** - Event Queues prevent the recycling of receive buffers. When the UM library transfers a received message to an event queue for later processing, it allocates (malloc) a new message receive buffer.
- **Message Object Retention** - Message retention prevents the recycling. For context-thread receive message callbacks, the act of retaining a message allocates (mallocs) a new message receive buffer.
- **Persistence** - For a persistent receiver, enabling receive buffer recycling will *reduce* dynamic memory usage (malloc/free), but does not eliminate it. Certain persistence-related features require the use of dynamic memory.
- **Packet Loss** - Applications typically use **Ordered Delivery**. When packets are lost, UM needs to internally retain newly received messages so that they can be delivered after the missing messages are retransmitted. This internal retention prevents the newly received message buffers from being recycled.
- **Message Fragmentation and Reassembly** - Large application messages must be split into smaller fragments and sent serially. The receiver must internally retain these fragments so that the original large message can be reassembled and delivered to the application. This internal retention prevents the fragment message buffers from being recycled.

Note that in spite of the restrictions that can prevent recycling of receive message buffers, UM dynamically takes advantage of recycling as much as it can. E.g. if there is a loss event which suppresses recycling while waiting for retransmission, once the gap is filled and pending messages are delivered, UM will once again be able to recycle its receive buffers.

Of specific interest for *persistent* receivers is the use of **Explicit Acknowledgments**, either to batch ACKs, or simply defer them. Instead of retaining the messages, which prevents message buffer recycling, you can extract the ACK information from a message and allow the return from the receiver callback to delete and recycle the message buffer without acknowledging it.

See **Object-free Explicit Acknowledgments** for details.

11.4 Single Receiving Thread

This feature optimizes the execution of UM receive-path code by converting certain thread-safe operations to more-efficient thread-unsafe operations. For example, certain bus-locked operations (e.g. atomic increment) are replaced by non-bus-locked equivalents (e.g. non-atomic increment). This can reduce the latency of delivering received messages to the application, but does so at the expense of thread safety.

This feature is often used in conjunction with the Context Lock Reduction feature.

The **transport_session_single_receiving_thread (context)** configuration option enables this feature.

Except as listed in the restrictions below, the Single Receiving Thread feature should be compatible with all other receive-side UM features.

11.4.1 Single Receiving Thread Restrictions

It is very important for applications using this feature to be designed within certain restrictions.

- **Threading** - The intended use case is for each received message to be fully processed by the UM thread that delivers the message to the application. Note that the Transport Services Provider (XSP) feature *is* compatible with the Single Receiving Thread feature.
- **No Event Queues** - Event queues cannot be used with Single Receiving Thread.
- **Message Object Retention** - Most traditional uses of message retention are related to giving a message to an alternate thread for processing. This is not compatible with Single Receiving Thread feature.

  However, there are some use cases where message retention is viable when used with Single Receiving Thread: when a message must be held for *future* processing, and that processing will be done by the same thread.

  For example, a persistent application might use **Explicit Acknowledgments** to delay message acknowledgement until the application completes a handshake with a remote service. As long as it is the same thread which initially receives and retains the message as that which completes the explicit acknowledgement of the message, it is supported to use message retain / message delete.

  Note

  > If the Transport Services Provider (XSP) feature is used, care must be taken to ensure that the same XSP thread is used to perform all processing for a received message. I.e. a different XSP or the main context may not be used to complete processing on a deferred retained message. For example, a user-scheduled timer event will be delivered using the main context thread, and therefore cannot complete processing of a retained message.

- **Transport Type** - The Single Receiving Thread feature does not enhance the operation of Broker or S↩MX transport types. These transport types use somewhat different internal buffer handling. Note that these transport types are technically compatible with the Single Receiving Thread feature, they just don't benefit from it.

11.5 Extended Context Process Events

Most developers of UM applications use a multi-threaded approach to their application design. For example, they typically have one or more application threads, and they create a UM context with **embedded mode**, which creates a separate context thread.

However, there is a model of application design in which a single thread is used for the entire application. In this case, the UM context must be created with Sequential Mode and the application must regularly call the UM event processor API, usually with the `msec` parameter set to zero. In this design, there is no possibility that application code, UM API code, and/or UM context code will be executing concurrently.

The **lbm_context_process_events_ex()** API allows the application to enable specialized optimizations. (For Java and .NET use the context object's "`processEvents()`" method with 2 or more input parameters. See **com↩ ::latencybusters::lbm::LBMContext::processEvents**.)

11.5.1 Context Lock Reduction

The application can improve performance by suppressing the taking of certain mutex locks within the UM context processing code. This can reduce the latency of delivering received messages to the application, but does so at the expense of thread safety.

This feature is often used in conjunction with the Single Receiving Thread feature.

Warning

> It is very important for the application to ensure that UM code related to a given context cannot be executed concurrently by multiple threads when this feature is used. This includes UM object creation and send-path API functions. I.e. the application may not call a UM message send API by one thread while another thread is calling **lbm_context_process_events_ex()**. However, it is permissible for a context thread callback to call a UM message send API, within the restrictions of the send API being used.

To enable this feature, call **lbm_context_process_events_ex()**, passing in the **lbm_process_events_info_t** structure with the **LBM_PROC_EVENT_EX_FLAG_NO_MAIN_LOOP_MUTEX** bit set in the `flags` field. (Sequential Mode is required for this feature.)

11.5.2 Context Lock Reduction Restrictions

It is very important for applications using this feature to be designed within certain restrictions.

- **Threading** - It is critical that Context Lock Reduction be used *only* if Sequential Mode is used and there is no possibility of concurrent execution of UM code for a given context.

  It is further strongly advised that the same thread be used for all UM execution within a given context. I.e. it is not guaranteed to be safe if the application has multiple threads which can operate on a context, even if the application guarantees that only one thread at a time will execute the UM code.

  Note that if an application maintains two contexts, it is acceptable for a different thread to be used to operate on each context. However, it is not supported to pass UM objects between the threads.

- **No Transport Services Provider (XSP)** - The Context Lock Reduction feature is not compatible with XSP.

- **No Event Queues** - Event queues cannot be used with Context Lock Reduction.

- **No SMX or DBL** - Context Lock Reduction is not compatible with SMX or DBL transports. This is because these transports create independent threads to monitor their respective transport types.

- **Transport LBT-IPC** - Context Lock Reduction was not designed with the IPC transport in mind. By default, IPC creates an independent thread to monitor the shared memory, which is not compatible with Context Lock Reduction. However, in principle, it is possible to specify that the IPC receiver should use sequential mode (see **transport_lbtipc_receiver_operational_mode (context)**), and then write your application to use the same thread to call the context and IPC event processing APIs. However, be aware that the IPC event processing API does not have an extended form, so IPC will simply continue to take the locks it is designed to take.

- **Message Object Retention** - Most traditional uses of Message Object Retention are related to handing a message to an alternate thread for processing. This is not compatible with Context Lock Reduction because the alternate thread is responsible for deleting the message when it is done. This represents two threads making API calls for the same context, which is not allowed for the Context Lock Reduction feature.

  However, there are some use cases where message retention is viable when used with Context Lock Reduction: when a message must be held for *future* processing, and that processing will be done by the same thread.

  For example, a persistent application might use **Explicit Acknowledgments** to delay message acknowledgement until the application completes a handshake with a remote service. As long as it is the same thread which initially receives and retains the message as that which completes the explicit acknowledgement of the message, it is supported to use message retain / message delete.

- **No LBM_SRC_BLOCK** - All forms of UM send message must be done non-blocking (i.e. with **LBM_SR↵C_NONBLOCK**). This is because of the way UM blocks calls that cannot be completed; the context thread explicitly wakes up the blocked call when appropriate. But if the same thread is being used to run the context (via the process events API) and also sending messages, a blocked send call will never be woken up.

11.5.3 Gettimeofday Reduction

UM's main context loop calls gettimeofday() in strategic places to ensure that its internal timers are processed correctly. However, there is a "polling" model of application design in which Sequential Mode is enabled and the context event processing API is called in a fast loop with the `msec` parameter set to zero. This results in the internal context call to gettimeofday() to happen unnecessarily frequently.

A polling application can improve performance by suppressing the internal context calls to gettimeofday(). This can reduce the latency of delivering received messages to the application.

To enable this feature, call **lbm_context_process_events_ex()**, passing in the **lbm_process_events_info_t** structure with the **LBM_PROC_EVENT_EX_FLAG_USER_TIME** bit set in the `flags` field. In addition, the application must set the `time_val` field in **lbm_process_events_info_t** with the value returned by gettimeofday(). (Sequential Mode is required for this feature.)

Note

> The internal UM timers generally use millisecond precision. Users of the gettimeofday() reduction feature typically design their application to fetch a new value for `time_val` only a few times per millisecond.

11.5.4 Gettimeofday Reduction Restrictions

- **Monotonically Increasing Time** - The application is responsible for ensuring that each call to **lbm_context↵_process_events_ex()** has a `time_val` field value which is greater than or equal to the previous `time↵_val`.

- **`"msec"` must be 0** - You must call **lbm_context_process_events_ex()** with the `"msec"` parameter set to zero. You can't tell **lbm_context_process_events_ex()** to loop for a period of time, and also tell it not to call gettimeofday(); UM won't see the passage of time.

11.6 Receive Multiple Datagrams

A UM receiver for UDP-based protocols normally retrieves a single UDP datagram from the socket with each socket read. Setting **multiple_receive_maximum_datagrams (context)** to a value greater than zero directs UM to retrieve up to that many datagrams with each socket read. When receive socket buffers accumulate multiple messages, this feature improves CPU efficiency, which reduces the probability of loss, and also reduces total latency for those buffered datagrams. Note that UM does not need to wait for that many datagrams to be received before processing them; if fewer datagrams are in the socket's receive buffer, only the available datagrams are retrieved.

The **multiple_receive_maximum_datagrams (context)** configuration option defaults to 0 so as to retain previous behavior, but users are encouraged to set this to a value between 2 and 10. (Having too large a value during a period of overload can allow starvation of low-rate Transport Sessions by high-rate Transport Sessions.)

Note that in addition to increasing efficiency, setting **multiple_receive_maximum_datagrams (context)** greater than one can produce changes in the dynamic behavior across multiple sockets. For example, let's say that a receiver is subscribed to two Transport Sessions, A and B. Let's further say that Transport Session A is sending message relatively quickly and has built up several datagrams in its socket buffer. But in this scenario, B is sending slowly. If **multiple_receive_maximum_datagrams (context)** is zero or one, the two sockets will compete equally for UM's attention. I.e. B's socket will still have a chance to be read after each A datagram is read and processed.

However, if **multiple_receive_maximum_datagrams (context)** is 10, then UM can process up to 10 of A's messages before giving B a chance to be read. This is desirable if low message latency is equally important across all Transport Sessions; the efficiency improvement derived by retrieving multiple datagrams with each read operation results in lower overall latency. However, if different transport sessions' data have different priorities in terms of latency, then processing 10 messages of a low priority transport session can unnecessarily delay processing of messages from a higher priority transport session.

In this case, the Transport Services Provider (XSP) feature can be used to prioritize different transport sessions differently and prevent low-priority messages from delaying high-priority messages.

Note that UM versions prior to 6.13 could see occasional increases in latency outliers when this feature was used. As of UM version 6.13, those outliers have been fixed (see **bug10726**).

11.6.1 Receive Multiple Datagrams Compatibility

The Receive Multiple Datagrams feature modifies the behavior of the UDP-based transport protocols: LBT-RM and LBT-RU.

(Note: prior to UM version 6.13, the Receive Multiple Datagrams feature also affected MIM and UDP-based Topic Resolution. But this could introduced undesired latencies, so starting with UM 6.13, MIM and Topic Resolution no longer use Receive Multiple Datagrams.)

11.6.2 Receive Multiple Datagrams Restrictions

The Receive Multiple Datagrams feature does not affect the following UM features:

- Non-UDP Transport Protocols (TCP, IPC, SMX).
- MIM (as of UM version 6.13).
- UDP-based Topic Resolution (as of UM version 6.13).
- All TCP-based features (Unicast Immediate Message, Late Join, Persistent Store Recovery, UM Response messages).
- Non-Linux.

- Linux prior to kernel version 2.6.33, and glibc in version 2.12 (released in May, 2010).

11.7 Transport Demultiplexer Table Size

A UM Transport Session can have multiple sources (topics) mapped to it. For example, if a publishing application creates two sources with the same multicast address and destination port, both sources will be carried on the same transport session. A receiver joined to that transport session must separate (demultiplex) the topics, either for delivery to the proper receiver callback, or for discarding.

The demultiplexing of the topics is managed by a hash table (not the same kind of hash table that manages the Topic Resolution cache). As a result, the processing of received messages can be made more efficient by optimally sizing the hash table. This is done using the configuration option **transport_demux_tablesz (receiver)**.

Unlike many hash tables, the transport demultiplexer needs to have a number of buckets which is a power of two. The demultiplexing code will be most efficient if the number of buckets is equal to or greater than the number of sources created on the transport session. In that case, the hash function is "perfect", which is to say that there will never be any collisions. Note that if the number of buckets is smaller than the number of sources, the collision resolution process is O(log N) where N is the length of the collision chain.

The only disadvantage of increasing the size of the hash table is memory usage (each bucket is 16 bytes on 64-bit architectures). Having a larger than optimal table does not make performance worse.

Note that if the number of sources is small, only a small degree of efficiency improvement results from optimally sizing the hash table.

11.8 Smart Sources

The normal **lbm_src_send()** function (and its Java and .NET equivalents) are very flexible and support the full range of UM's rich feature set. To provide this level of capability, it is necessary to make use of dynamic (malloc/free) memory, and critical section locking (mutex) in the send path. While modern memory managers and thread locks are very efficient, they do introduce some degree of variability of execution time, leading to latency outliers (jitter) potentially in the millisecond range.

For applications which require even higher speed and very consistent timing, and are able to run within certain constraints, UM has an alternate send feature called Smart Source. This is a highly-optimized send path with no dynamic memory operations or locking; all allocations are done at source creation time, and lockless algorithms are used throughout. To achieve these improvements, Smart Source imposes a number of restrictions (see Smart Sources Restrictions).

The Smart Source feature provides the greatest benefit when used in conjunction with a kernel bypass network driver.

Note

> the Smart Source feature is *not* the same thing as the Zero-Copy Send API feature; see Comparison of Zero Copy and Smart Sources.

One design feature that is central to Smart Sources is the pre-allocation of a fixed number of carefully-sized buffers during source creation. This allows deterministic algorithms to be used for the management of message buffers throughout the send process. To gain the greatest benefit from Smart Sources, the application builds its outgoing messages directly in one of the pre-allocated buffers and submits the buffer to be sent.

To use Smart Sources, a user application typically performs the following steps:

1. Create a context with **lbm_context_create()**, as normal.
2. Create the topic object and the Smart Source with **lbm_src_topic_alloc()** and **lbm_ssrc_create()**, respectively. Use Smart Sources Configuration to pre-allocate the desired number of buffers.
3. Get the desired number of messages buffers with **lbm_ssrc_buff_get()** and initialize them if desired. The application typically constructs outgoing messages directly in these buffers for transmission.
4. Send messages with **lbm_ssrc_send_ex()**. The buffers gotten in the previous step must be used.
5. While most applications manage the message buffers internally, it is also possible to give the buffers back to UM with **lbm_ssrc_buff_put()**, and then getting them again for subsequent sends. Getting and putting messages buffers can simplify application design at the expense of extra overhead.
6. To clean up, delete the Smart Source with **lbm_ssrc_delete()**. It is not necessary to "put" the message buffers back to UM; they will be freed automatically when the Smart Source is deleted.

For details, see the example applications **Example lbmssrc.c** or **Example lbmssrc.java**.

Warning

> To avoid the overhead of locking, the Smart Source API functions are not thread-safe. Applications must be written to avoid concurrent calls. In particular, the application is restricted to sending messages on a given Transport Session with one thread. If Smart Source Defensive Checks are enabled, the first call to send a message on a newly-created Transport Session captures the ID of the calling thread. Subsequently, only that thread is allowed to call send for Smart Sources on that Transport Session. For applications which have multiple sending threads, Smart Source topics must be mapped to Transport Sessions carefully such that all of the topics on a given Transport Session are managed by the same sending thread.

Note

> There are no special requirements on the receive side when using Smart Sources. Normal receiving code is used.

11.8.1 Smart Source Message Buffers

When a Smart Source is created, UM pre-allocates a set of user buffers according to the configuration options **smart_src_max_message_length (source)** and **smart_src_user_buffer_count (source)**.

As of UM version 6.12, Smart Source supports UM fragmentation. Which is to say that messages larger than the transport's Datagram Max Sizes can be sent, which the Smart Source will split into multiple datagrams.

For example, an application can configure **smart_src_max_message_length (source)** to be 2000, while the datagram max size is set to 1500 (network MTU size). During operation, the application might send a 500-byte message. This will not require any fragmentation; the message is sent in a single network packet. However, when the application sends a 2000-byte message, the Smart Source will split it into two datagrams. This avoids IP fragmentation. The precise sizes of those datagrams will depend on the space reserved for headers, and is subject to change with different versions of UM.

Another feature available as of UM version 6.12 is the user-specified buffer. This allows an application to send messages larger than the configured **smart_src_max_message_length (source)**. Instead of building the message in a pre-allocated Smart Source buffer, the application must allocate and manage its own user-supplied buffer. To use this feature, the application supplies both a pre-allocated buffer and a user-supplied buffer. The Smart Source will use the supplied pre-allocated buffer as a "work area" for building the datagram with proper headers, and use the user-supplied buffer for message content.

For example to use the buffer `"ubuffer"`, you simply set the **LBM_SSRC_SEND_EX_FLAG_USER_SUPP↩ LIED_BUFFER** flag and the `usr_supplied_buffer` field in the **lbm_ssrc_send_ex_info_t** passed to the **lbm_ssrc_send_ex()** API function, as shown below:

```
char *ubuffer = malloc(65536);  /* Large user-supplied buffer. */
lbm_ssrc_send_ex_info_t info;
info.flags = 0;
char *ss_buffer = NULL;  /* Smart Source pre-allocated buffer. */
...
lbm_ssrc_buff_get(ssrc, &ss_buffer, 0);  /* Get Smart Source pre-alloc buff. */
...
/* Application puts message data into ubuffer. */
info.flags |= LBM_SSRC_SEND_EX_FLAG_USER_SUPPLIED_BUFFER;
info.usr_supplied_buffer = ubuffer;
lbm_ssrc_send_ex(ssrc, ss_buffer, message_len, 0, &info);
```

Note that the Smart Source pre-allocated buffer `ss_buffer` also has to be passed in.

Also note that sending messages with the user-supplied message buffer is slightly less CPU efficient than using the pre-allocated buffers. But making pre-allocated buffers larger to accommodate occasional large messages can be very wasteful of memory, depending on the counts of user buffers, transmission window buffers, and retention buffers.

UM Fragment Sizes

A traditional source will split application messages into "N" fragments when those messages (plus worst-case header) are greater than the Datagram Max Sizes. The size of the first "N-1" fragments will be (approximately) the datagram max size.

With Smart Sources, fragmentation is done somewhat differently. Consider as an example a configuration with a datagram max size of 8192 and a Smart Source max message length of 2000. No UM message fragmentation will happen when the application uses the Smart Source pre-allocated buffers to build outgoing messages. However, if a user-supplied buffer is used, the user can send arbitrarily large application message, and the Smart Source will split the message into "N" fragments. But those fragments will be limited in size to the Smart Source max message length of 2000 bytes of application data (plus additional bytes for headers).

This can lead to unexpected inefficiencies. Continuing the above example, suppose case the application sends a 6000-byte message. The Smart Source will spit it into three 2000-byte datagrams. The underlying IP stack will perform IP fragmentation and send each datagram as two packets of 1500 and 500 bytes respectively, for a total of 6 packets. Whereas if the Smart Source max message length were set to 1500, then the message would be split into 4 fragments of 1500 bytes each, and each fragment would fit in a single packet, for a total of 4 packets. (The calculations above were simplified for clarity, but are not accurate because they do not take into consideration headers.)

When a kernel bypass network driver is being used, users will sometimes set the datagram max size to approximately an MTU. In that case, it could easily happen that the Smart Source pre-allocated buffers are *larger* than the datagram max size. In that case, the Smart Source will behave more like a traditional source, splitting the application message into datagrams of (approximately) datagram max size fragments. See Datagram Max Size and Network MTU.

11.8.2 Smart Sources and Memory Management

As of UM 6.11, there are new C APIs that give the application greater control over the allocation of memory when Smart Sources are being created. Since creation of a Smart Source pre-allocates buffers used for application message data as well as internal retransmission buffers, an application can override the stock malloc/free to ensure, for example, that memory is local to the CPU core that will be sending messages.

When the application is ready to create the Smart Source, it should set up the configuration option **mem_mgt_↩ callbacks (source)**, which uses the **lbm_mem_mgt_callbacks_t** structure to specify application callback functions.

11.8.3 Smart Sources Configuration

The following configuration options are used to control the creation and operation of Smart Sources:

- **smart_src_max_message_length (source)** - should be set to the maximum expected size for messages sent to on the source.
- **smart_src_user_buffer_count (source)** - number of buffers to be pre-created at Smart Source create time. Deleting a Smart Source also frees these buffers, so applications must not access these buffers after their corresponding Smart Source is deleted.
- **smart_src_retention_buffer_count (source)** - enables Late Join and Off-Transport Recovery (OTR) functionality. Takes the place of the normal late join / OTR options "retransmit_retention_∗". (On the receive side, the normal late join options apply.)
- **transport_lbtrm_smart_src_transmission_window_buffer_count (source)** - size of the LBT-RM transmission window. Takes the place of the normal window options "transport_lbtrm_transmission_window_∗".
- **transport_lbtru_smart_src_transmission_window_buffer_count (source)** - size of the LBT-RU transmission window. Takes the place of the normal window options "transport_lbtru_transmission_window_∗".
- **smart_src_enable_spectrum_channel (source)** - should be set if Spectrum channels will be used.
  See Smart Sources and Spectrum.
- **smart_src_message_property_int_count (source)** - should be set if Message Properties will be used.
  See Smart Sources and Message Properties.

The option **smart_src_max_message_length (source)** is used to size the window transmission buffers. This means that the first Smart Source created on the session defines the maximum possible size of user messages for all Smart Sources on the Transport Session. It is not legal to create a subsequent Smart Source on the same Transport Session that has a larger **smart_src_max_message_length (source)**, although smaller values are permissible.

11.8.4 Smart Source Defensive Checks

Ultra Messaging generally includes defensive checks in API functions to verify validity of input parameters. In support of faster operation, deep defensive checks for Smart Sources are optional, and are disabled by default. Users should enable them during application development, and can leave them disabled for production.

To enable deep Smart Source defensive checks, set the environment variable **LBM_SMART_SOURCE_CHECK** to the numeric sum of desired values. Hexadecimal values may be supplied with the "0x" prefix. Each value enables a class of defensive checking:

Numeric Value	Deep Check
1	Send argument checking
2	Thread checking
4	User buffer pointer checking
8	User buffer structure checking
16, 0x10	user message length checking
32, 0x20	application header checking, including Spectrum and Message Properties.
64, 0x40	null check for User Supplied Buffer (see Smart Source Message Buffers)

To enable all checking, set the environment variable **LBM_SMART_SOURCE_CHECK** to "0xffffffff".

11.8.5 Smart Sources Restrictions

- **Linux and Windows 64-bit Only** - Smart Sources is only supported on the 64-bit Linux and 64-bit Windows platforms, C and Java APIs.
- **LBT-RM And LBT-RU Sources Only** - Smart Sources can only be created with the LBT-RM and LBT-RU transport types. Smart Sources are not compatible with the UM features Multicast Immediate Messaging, Unicast Immediate Messaging, or sending responses with Request/Response.
- **Persistence** - As of UM 6.11, Smart Sources support Persistence, but with some restrictions. See **Smart Sources and Persistence** for details.
- **Spectrum** - As of UM 6.11, Smart Sources support Spectrum, but with some API changes. See Smart Sources and Spectrum for details.
- **Single-threaded Only** - It is the application's responsibility to serialize calls to Smart Source APIs for a given Transport Session. Concurrent sends to different Transport Sessions are permitted.
- **No Application Headers** - Application Headers are not compatible with Smart Sources.
- **Limited Message Properties** - Message Properties may be included, but their use has restrictions. See Smart Source Message Properties Usage.
- **No Queuing** - Queuing is not currently supported, although support for ULB is a possibility in the future.
- **No Send Request for Java** - The Java API does not support sending UM Requests. (As of UM version 6.14, the C API does: **lbm_ssrc_send_request_ex()**).
- **No Data Rate Limit** - Smart Source data messages are not **rate limited**, although retransmissions are **rate limited**. Care must be taken in designing and provisioning systems to prevent overloading network and host equipment, and overrunning receivers.
- **No Hot Failover** - Smart Sources are not compatible with Hot Failover (HF).
- **No Batching** - Smart Sources are not compatible with Implicit Batching or Explicit Batching.

Note

> It is not permitted to mix Smart Source API calls with standard source API calls for a given Transport Session.

11.9 Zero-Copy Send API

This section introduces the use of the zero-copy send API for LBT-RM.

Note

> the Zero-Copy Send API feature is *not* the same thing as the Smart Sources feature; see Comparison of Zero Copy and Smart Sources.

The zero-copy send API modifies the **lbm_src_send()** function for sending messages such that the UM library does not copy the user's message data before handing the datagram to the socket layer. These changes reduce CPU overhead and provide a minor reduction in latency. The effects are more pronounced for larger user messages, within the restrictions outlined below.

Application code using the zero-copy send API must call **lbm_src_alloc_msg_buff()** to request a message buffer into which it will build its outgoing message. That function returns a message buffer pointer and also a separate buffer handle. When the application is ready to send the message, it must call **lbm_src_send()**, passing the buffer handle as the message (not the message buffer) and specify the LBM_MSG_BUFF_ALLOC send flag.

Once the message is sent, UM will process the buffer asynchronously. Therefore, the application must not make any further reference to either the buffer or the handle.

11.9.1 Zero-Copy Send Compatibility

The zero-copy send API is compatible with the following UM features:

- C language, Streaming, source-based publishing applications using LBT-RM.
- Messages sent with the zero-copy API can be received by any UM product or daemon. No special restrictions apply to receivers of messages sent with the zero-copy send API.
- Compatible with implicit batching and message flushing.
- Compatible with non-blocking sends and wakeup source event handling.
- Compatible with hardware timestamps (see section High-resolution Timestamps).
- Compatible with UD Acceleration.

11.9.2 Zero-Copy Restrictions

Due to the specialized nature of this feature, there are several restrictions in its use:

- **Languages**. Java and .NET are not supported at this time.
- **Transport LBT-RM only**. Sourced-based LBT-RM (multicast) only. Zero-copy sends are not compatible with LBT-RU, TCP, IPC, SMX, Unicast Immediate Messaging, or Multicast Immediate Messaging. Note that an application that uses zero-copy sends for certain sources may also have other sources configured for other transport types.
- **Applications only**. UM daemons (e.g. DRO, Stored, etc.) cannot be configured to use the zero-copy API.
- **Streaming only**. Zero-copy sends are not compatible with Persistence or Queuing. Note that an application that uses zero-copy sends for certain sources may also have other sources mapped to Persistence and/or queuing.
- **lbm_src_send() only**. zero-copy sends are not compatible with send APIs not supported: **lbm_src_sendv()**, **lbm_src_send_ex()**, **lbm_src_sendv_ex()**, **lbm_hf_src_send()**, **lbm_hf_src_sendv()**, **lbm_hf_src_send_ex()**, **lbm_hf_src_sendv_ex()**, **lbm_send_request()**, **lbm_send_request_ex()**, **lbm_send_response()**, **lbm_multicast_immediate_message()**, **lbm_multicast_immediate_request()**, **lbm_unicast_immediate_message()**, **lbm_unicast_immediate_request()**. Applications may still use these APIs, but not with the zero-copy send feature.
- **Send order**. It is recommended that zero-copy buffers be sent in the same order that they are allocated. A future version may require this.
- Late Join. not compatible with zero-copy sends. Note that an application that uses zero-copy sends on certain sources may also use late join on other sources.
- Request/Response. not not compatible with zero-copy sends.
- Message Metadata. Not compatible with Message Properties or Application Headers. Note that an application that uses zero-copy sends for messages without metadata may also send messages with metadata using other send APIs, even to the same source.

- Hot Failover (HF). not supported. Note that an application that uses zero-copy sends for certain sources may use hot failover for other sources.

- Explicit Batching. not compatible with zero-copy sends. Note that implicit batching is supported. Also note that an application that uses zero-copy sends for certain sources may use explicit batching for other sources.

- Message Fragmentation and Reassembly. not compatible with zero-copy sends. Messages sent zero-copy must fit within a single datagram, as defined by the LBT-RM Datagram Max Sizes. No special restrictions apply to IP fragmentation. Note that an application that uses zero-copy sends for single-datagram messages may also send multi-datagram messages using other send APIs, even to the same source.

11.10 Comparison of Zero Copy and Smart Sources

There are two UM features that are intended to reduce latency and jitter when sending messages:

- Smart Sources

- Zero-Copy Send API

These two features use different approaches to latency and jitter reduction, and are not compatible with each other. There are trade-offs explained below, and users seeking latency and/or jitter reduction will sometimes need to try both and empirically measure which is better for their use case.

The zero-copy send API removes a copy of the user's data buffer, as compared to a normal send. For small messages of a few hundred bytes, a malloc and a data copy represent a very small amount of time, so unless your messages are large, the absolute latency reduction is minimal.

The Smart Source has the advantage of eliminating all mallocs and frees from the send path. In addition, all thread locking is eliminated. This essentially removes all sources of jitter from the UM send path. Also, the Smart Source feature supports UM fragmentation, which zero-copy sends do not. However, because of the approach taken, sending to a Smart Source is somewhat more restrictive than sending with the zero-copy API.

In general, Informatica recommends Smart Sources to achieve the maximum reduction in jitter. For example, the zero-copy send API supports the use of batching to combine multiple messages into a single network datagram. Batching can be essential to achieve high throughputs. Some application designers may determine that the throughput advantages of zero-copy with batching outweigh the jitter advantages of Smart Sources.

See the sections Zero-Copy Send API and Smart Sources for details of their restrictions.

11.11 XSP Latency Reduction

A common source of latency outliers is when Topic Resolution packets are received at the same time that user data messages are received. The UM context thread might process those Topic Resolution packets before processing the user data messages.

By using the XSP feature, user data reception can be moved to a different thread than topic resolution reception. See Transport Services Provider (XSP) for details, paying careful attention to XSP Threading Considerations.

11.12 Receive-Side Batching

The receive-side batching feature can improve throughput of subscribers that use UM event queues and/or are written in Java.

See also source-side Message Batching.

UM event queues introduce a small overhead which adds latency and reduces maximum sustainable throughput. UM's Java API also introduces a small overhead to deliver received messages from the UM library. Both of these overheads can usually be mitigated with the receive-side batching feature.

This feature is enabled via the configuration option **delivery_control_message_batching (context)**.

The feature works by collecting receiver events into bundles. For an event queue, the bundle is sent to the dispatch thread as a single queue item. For Java, the bundle is passed to the Java language as a single object. In both cases, the overhead is amortized across the multiple messages in the bundle.

This feature is most effective when used along with the Receive Multiple Datagrams feature (Linux only), which will increase the bundle size automatically if the receiver falls behind. A publisher using Message Batching also increases the bundle size. Larger bundles means greater efficiency gains.

Warning

> Do not use receive-side batching for C or .NET programs that are not using event queues. The feature does not increase efficiency and can actually reduce it.
> This feature is not compatible with XSP. Users of XSP are typically trying to maximize the performance of UM, and probably should not be using event queues. For alternatives, please `contact Informatica Support`.

Note

> If you enable receive-side batching and you use an event queue that is in polling mode, using lbm_event_dispatch(evq, LBM_EVENT_QUEUE_POLL), then rather than dispatching exactly one event per call to lbm_event_dispatch, you may get multiple events dispatched with a single call.

11.13 Core Pinning

The Unix and Windows operating systems attempt to balance CPU utilization across all available CPU cores. They often do this without regard to the architectural design of the system hardware, which can introduce significant inefficiencies. For example, if a thread's execution migrates from one `NUMA node` to another, the code will frequently need to access memory located in the other NUMA zone, which happens over a slower memory interconnect.

Fortunately, Unix and Windows support pinning processes and threads to specific CPU cores. It is the user's responsibility to understand the host architecture sufficiently to know which cores are grouped into NUMA zones. Pinning a group of related threads to cores within the same NUMA zone is important to maintain high performance.

However, even letting the operating system migrate a thread from one core to another within a single NUMA zone has the side effect of invalidating the cache, which introduces latency. You get the best performance when each thread is pinned to its own core, with no other threads contending for that core. This approach obviously severely limits the number of threads that can run on a host.

UM does not have a general feature that pins threads to cores for applications. It is the user's responsibility to pin (set affinity) using the appropriate operating system APIs.

The Persistent Store allows the user to assign individual threads to specific cores. See **Store Thread Affinity** for details.

For the DRO, it is not possible to identify specific threads and assign them to individual cores. But the user can use the operating system's user interface to assign the entire DRO process to a group of cores known to be in the same NUMA zone.

11.14 Memory Latency Reduction

UM makes use of dynamic memory allocation/deallocation using malloc() and free(). The default memory allocator included with Linux and Windows can sometimes introduce latency outliers of multiple milliseconds. It is rare, but we have seen outliers as long as 10 milliseconds.

There are higher-performing allocators available, many of them open-source. For example, Hoard. There are many others.

A good commercial product is MicroQuill's SmartHeap. In fact, the Persistent Store is built and ships with SmartHeap. Note that licensing Ultra Messaging does not grant a license to the customer for general use of SmartHeap. Users who want to use SmartHeap in applications should contact MicroQuill directly.

None of these products can **guarantee** that there will never be millisecond-long latencies, but they can greatly reduce the frequency.

Chapter 12

Man Pages for SRS

TCP-based resolver services for UM messaging products are provided by SRS.

For more information on TCP-based TR, see TCP-Based Topic Resolution Details. For more information on Topic Resolution general, see Topic Resolution Description.

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

- SRS Man Page - Unix and Windows command-line interface.
- Srsds Man Page - Windows Service interface.

Note that these executables are not in the same "bin" directory as the platform native UM executables. Since it is a Java program, it has its own directory sub-tree, "SRS", with sub-directories "bin" and "lib". For example, the UMS SRS executables are under "UMS_6.16.1/SRS/bin".

However, SRS does use the underlying platform-specific UM library, so your proper paths should be set up.

12.1 SRS Man Page

Unix and Windows command-line interface.

```
Usage: SRS [options] [configfile]
Available options:
  -d, --dump              dump the user configuration to stdout and exit
  -D, --Debug=PATH:MASK   set debug PATH and MASK
  -h, --help              display this help message and exit
  -j, --java              print Java properties to the SRS log file (-j -j = print
      more Java properties)
  -v, --validate          validate config file and exit
  -x, --xsd               dump the configuration XSD to stdout and exit
```

Description

The SRS command runs the Stateful Resolver Service (SRS). It can be run interactively from a shell or command prompt, or from a script or batch file. (For use as a Windows Service, see Srsds Man Page.)

The **"configfile"** parameter is optional. If supplied, it specifies the file path for the SRS's XML configuration file. if omitted, the SRS defaults all configurable options. See SRS Configuration File for configuration details.

The **"-D"** option sets enables debugging output. This output is intended primarily for Informatica Support, not end-users.

The **"-j"** option prints Java properties to the SRS log file. It can be repeated ('-j -j') to increase the output. This output is intended primarily for Informatica Support, not end-users.

The **"-d"** option dumps (prints) to standard out the full SRS configuration. After printing, the SRS exits. (Note, this is different from other UM daemons in which "-d" dumps the daemon's DTD. But the SRS does not use a DTD, it uses an XSD. See the "-x" option below.)

The **"-x"** option prints the XSD which is used to validate the configuration file. After printing, the SRS exits.

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

```
SRS -v /um/srs_cfg.xml
```

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

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

Exit Status

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

12.2 Srsds Man Page

Windows Service interface.

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

Note

> In the descriptions below, three different log files are referenced: "service log", "process log", and "SRS log". It is important that all three of these be specified, and that they be separate files. In normal use, only the "SRS log" will be written to; the other two are only necessary to record unusual error conditions.

```
Usage: srsds [options] srs_cfgfile_name
Available options:
  -h, --help                display this help and exit
  -l, --service-log=FILE    set a logfile name for the service log.
  -p, --process-log=FILE    set a logfile name for the srs process output.
  -s, --service=request     Install, remove or add a configuration file.
                            Examples: '-s install' to install the service with no
                                config file
                              '-s install cfgfile.xml' to install the service
                                  with a configuration file of cfgfile.xml
                              '-s remove' to remove the service
                              '-s config cfg2.xml' to change or add a
                                  configuration file
  -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
  -E, --env_var_file          update/set the environment Variable File
```

Description

The `srsds` 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 SRS Service. See **Configure the Windows Service**.
- Second, it provides Windows with the SRS daemon executable to run as a Service.

The **"srs_cfgfile_name"** parameter specifies the file path for the SRS's XML configuration file. It is supplied in conjunction with the "-s config" option (see below). See SRS Configuration File for configuration details.

The **"-l"** option specifies a "service" log file path, which is saved in the Windows registry and subsequently by the Windows Service. Under normal circumstances, this log file will never be written to. It will be written if the SRS is unsuccessful in starting up as a service. (The normal SRS log file is configured differently, using the <log> element in the configuration file.)

The **"-p"** option specifies a "process" log file path, which is saved in the Windows registry and subsequently by the Windows Service. Under normal circumstances, this log file will never be written to. It will be written if the Java JVM writes to standard out or standard error, of if the SRS is not able to write to its configured log file.

Warning

> By default, the SRS's configuration file uses <log type="console">. In this case, normal SRS logs are written to standard out and will be captured in the "process" log file. **This is not recommended as it leads to unbounded growth of the log file.** Users should set <log type="file"> and related attributes to control the log file sizes.

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

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

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

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

Exit Status

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

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.

Chapter 13

SRS Configuration File

For general information about TCP-based topic resolution, see TCP-Based Topic Resolution Details.

The SRS configuration file must start with this line:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

After that, the '<**um-srs**>' element contains the rest of the configuration.

The SRS configuration does not support the XInclude feature.

Here is a sample short configuration:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <log type="file" frequency="hourly" size="10" max-history="10"
         total-size-cap="10000" compression="zip">SRS.log</log>
    <pid-file>SRS.pid</pid-file>
  </daemon>
  <srs>
    <interface>10.12.34.56</interface>
    <port>27000</port>
    <state-lifetime>3600</state-lifetime>
  </srs>
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <default>2000</default>
      <config-opts>20000</config-opts>
      <internal-config-opts>0</internal-config-opts>
    </publishing-interval>
    <lbm-attributes>
      <option scope="context" name="operational_mode" value="embedded" />
      <option scope="context" name="mim_incoming_address" value="0.0.0.0" />
      <option scope="context" name="transport_tcp_port_low" value="14381" />
      <option scope="context" name="transport_tcp_port_high" value="15381" />
      <option scope="source" name="transport_tcp_interface" value="10.29.3.0/24" />
    </lbm-attributes>
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
</um-srs>
```

Attention

> It is important to configure the SRS with SRS Element "<interface>", even if the SRS is running on a single-homed host (i.e. one network connection).

When the daemon monitor is enabled, the first two lbm attribute option setting are recommended for conserving resources.

13.1 SRS Configuration Elements

13.1.1 SRS Element "<um-srs>"

Container element which holds the SRS configuration. Also defines the version of the configuration format used by the file.

- **Children:** <daemon>, <srs>, <debug-monitor>, <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
version	Version number of user's configuration file.	nonEmptyString	"1.0"

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  ...
</um-srs>
```

13.1.2 SRS Element "<daemon-monitor>"

Contains elements which configure the SRS monitoring capability. This feature is used to monitor the SRS's health and performance. It can also be useful to monitor activity in the entire Topic Resolution Domain.

Note that the "topic" attribute *must* be supplied, otherwise no statistics will be published. Also note that if using the protobuf format, other UM components default to the topic "/29west/statistics".

See child elements for details.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <um-srs>
- **Children:** <ping-interval>, <publishing-interval>, <lbm-attributes>, <publish-connection-events>, <remote-snapshot-request>, <remote-config-changes-request>, <monitor-format>

XML Attributes:

Attribute	Description	Valid Values	Default Value
topic	Set the name of the topic on which the SRS publishes its daemon stats.	nonEmptyString	(no default; must be specified)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.3 SRS Element "<monitor-format>"

Specifies the desired format of monitoring data. For use with the **Monitoring Collector Service (MCS)**, use the format "pb" (proto buff).

Valid values are "pb" (recommended) or "json" (deprecated). See SRS Monitoring.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <daemon-monitor>
- **Default Value:** json

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    ...
    <monitor-format>pb</monitor-format>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

13.1.4 SRS Element "<remote-config-changes-request>"

Controls whether the SRS daemon monitor accepts Request Type: SET_PUBLISHING_INTERVAL. This allows remote monitoring applications to change operational parameters of the SRS Daemon Monitoring feature.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
allow	Enable or disable this function.	**"true"** - Enabled **"false"** - Disabled	**"false"**

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <remote-config-changes-request allow="true"/>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.5 SRS Element "<remote-snapshot-request>"

Controls whether the SRS daemon monitor accepts Request Type: REPORT_MONITOR_INFO. This allows remote monitoring applications to trigger immediate publishing of monitoring data.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
allow	Enable or disable this function.	**"true"** - Enabled **"false"** - Disabled	**"false"**

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <remote-snapshot-request allow="true"/>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.6 SRS Element "<publish-connection-events>"

Controls whether the SRS reports connection-oriented events from UM contexts as part of the daemon stats. See SRS Daemon Statistics for more information.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
allow	Enable or disable this function.	**"true"** - Enabled **"false"** - Disabled	**"false"**

Example: (SRS includes connection-oriented events in monitoring stats)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publish-connection-events allow="true"/>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.7 SRS Element "<lbm-attributes>"

Container element containing any number of <option> elements. Each <option> element supplies an LBM configuration option to the UM context that the SRS creates to publish daemon stats. Any number of <option> elements can be supplied in the <lbm-attributes> container element.

See SRS Daemon Statistics for more information on daemon statistics.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <daemon-monitor>
- **Children:** <option>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <lbm-attributes>
      ...
    </lbm-attributes>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.8 SRS Element "<option>"

Supplies an LBM configuration option to the UM context that the SRS creates to publish daemon stats. Any number of <option> elements can be supplied in the <lbm-attributes> container element.

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

See SRS Daemon Statistics for more information on daemon statistics.

- **Cardinality** (number of times element can be supplied): 0 .. unbounded
- **Parent:** <lbm-attributes>

XML Attributes:

Attribute	Description	Valid Values	Default Value
scope	Scope for the LBM configuration option being set. One of: **"context"** **"source"** **"receiver"** (The normal LBM scopes `wildcard_receiver`, `event_queue`, and `hfx` are not applicable to the SRS monitor context.)	nonEmptyString	(no default; must be specified)
name	Name of LBM configuration option being set.	nonEmptyString	(no default; must be specified)
value	Value of LBM configuration option being set.	nonEmptyString	(no default; must be specified)

Example: (SRS publishes monitoring stats using LBT-RU transport)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <lbm-attributes>
      <option scope="context" name="operational_mode" value="embedded" />
      <option scope="context" name="mim_incoming_address" value="0.0.0.0" />
      <option scope="context" name="transport_tcp_port_low" value="14381" />
      <option scope="context" name="transport_tcp_port_high" value="15381" />
      <option scope="source" name="transport_tcp_interface" value="10.29.3.0/24" />
      ...
    </lbm-attributes>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

The first two option settings above are recommended for conserving resources.

13.1.9 SRS Element "<publishing-interval>"

Set how often the SRS publishes its daemon stats. See SRS Daemon Statistics for more information. The child elements set the intervals for each class of monitoring data. For any class of data omitted, the <default> element sets the interval.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <daemon-monitor>
- **Children:** <default>, <srs-stats>, <um-client-stats>, <connection-events>, <srs-error-stats>, <um-client-error-stats>, <config-opts>, <internal-config-opts>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.10 SRS Element "<internal-config-opts>"

Sets how often (in milliseconds) the SRS publishes certain internal configuration data. These data are primarily of interest to Informatica Support. The value zero disables publishing that class of daemon stats.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** 10

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <internal-config-opts>10000</internal-config-opts>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.11 SRS Element "<config-opts>"

Sets how often (in milliseconds) the SRS publishes its configuration data. The special value zero disables publishing that class of daemon stats.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <config-opts>10000</config-opts>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.12 SRS Element "<um-client-error-stats>"

Sets how often (in milliseconds) the SRS publishes statistics related to internal client-facing software errors. These statistics are primarily of interest to Informatica Support. The special value zero disables publishing that class of daemon stats.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <um-client-error-stats>10000</um-client-error-stats>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.13 SRS Element "<srs-error-stats>"

Sets how often (in milliseconds) the SRS publishes statistics related to internal SRS software errors. These statistics are primarily of interest to Informatica Support. The value zero disables publishing that class of daemon stats.

Valid values: 0, 200 - 7776000000

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <srs-error-stats>10000</srs-error-stats>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.14 SRS Element "<connection-events>"

Sets how often (in milliseconds) the SRS publishes client connect and disconnect events. The value zero disables publishing that class of daemon stats.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <connection-events>10000</connection-events>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.15 SRS Element "<um-client-stats>"

Sets how often (in milliseconds) the SRS publishes statistics related to Topic Resolution clients. The special value zero disables publishing that class of daemon stats.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <um-client-stats>10000</um-client-stats>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.16 SRS Element "<srs-stats>"

Sets how often (in milliseconds) the SRS publishes internal SRS operational statistics. The special value zero disables publishing that class of daemon stats.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
```

```
      <srs-stats>10000</srs-stats>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.17 SRS Element "<default>"

Sets how often (in milliseconds) the SRS publishes those classes of daemon stats which are not explicitly set by other elements. The special value zero disables publishing that class of daemon stats. See SRS Element "<publishing-interval>" for the classes.

Valid range: 0, 200 .. 7776000000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** 0

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <publishing-interval>
      <default>10000</default>
      ...
    </publishing-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.18 SRS Element "<ping-interval>"

Controls the period (in milliseconds) at which SRS internal statistics are sampled and made available. The special value zero disables this sampling.

The ping-interval control is available under the <daemon-monitor> element and the <debug-monitor> element. Unless otherwise directed by Informatica Support, only the <daemon-monitor> instance is of interest to users.

Note that there can only be one ping interval across both functions. A <daemon-monitor> setting for ping-interval will override a <debug-monitor> setting.

See SRS Monitoring.

Valid range: 100 .. 3600000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <debug-monitor>, <daemon-monitor>
- **Default Value:** 60000 (1 minute)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="/29west/statistics">
    <ping-interval>10000</ping-interval>
    ...
    <monitor-format>pb</monitor-format>
  </daemon-monitor>
  ...
</um-srs>
```

13.1.19 SRS Element "<debug-monitor>"

Contains elements which configure the optional web-based debug monitor for the SRS. The debug monitor is primarily for use by Informatica support, and is not intended for end users. Unless otherwise instructed by Informatica support, users should not enable the debug monitor.

Omit this element to disable the debug monitor.

See Webmon Security for important security information.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <um-srs>
- **Children:** <interface>, <port>, <ping-interval>, <enabled>

This is NOT related to monitoring the SRS health and performance. See <daemon-monitor>.

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <debug-monitor>
    ...
  </debug-monitor>
  ...
</um-srs>
```

13.1.20 SRS Element "<enabled>"

Controls whether the debug monitor is active. Can be set to `true` or `false`. See <debug-monitor>.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <debug-monitor>
- **Default Value:** false

Example: (disable debug-monitor explicitly)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <debug-monitor>
    <enabled>false</enabled>
    ...
  </debug-monitor>
  ...
</um-srs>
```

13.1.21 SRS Element "<port>"

Supplies network port to bind the socket required by the parent element. This is the port that a UM context should use when TCP-based TR is configured with the option **resolver_service (context)**. The value contained within the <port>...</port> is an integer.

Valid range: 0 .. 65535

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <debug-monitor>, <srs>
- **Default Value:** 27000

Example: (UM clients use port 12000 with **resolver_service (context)** option)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <port>12000</port>
    ...
  </srs>
  ...
</um-srs>
```

13.1.22 SRS Element "<interface>"

Specifies the network interface to bind the socket required by the parent element.

Attention

> It is important to configure the SRS with SRS Element "<interface>", even if the SRS is running on a single-homed host (i.e. one network connection).

For the <srs> element, this is the IP address that a UM context should use when TCP-based TR is configured with the option **resolver_service (context)**. The value contained within the <interface>...</interface> can be a fully-qualified dotted-decimal IP address or a DNS host name.

Warning

> Unlike UM library configurations, the SRS configuration does not support CIDR specification of an IP network to match an interface by network number. This interface specification must include the host number.

For the <debug-monitor> element, this is the host for the URL that a web browser should use to display the debug monitor page.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <debug-monitor>, <srs>
- **Default Value:** localhost

Example: (UM clients use 10.12.34.56 with **resolver_service (context)** option)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <interface>10.12.34.56</interface>
    ...
  </srs>
  ...
</um-srs>
```

13.1.23 SRS Element "<srs>"

Defines network and operational settings of the SRS service.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <um-srs>
- **Children:** <interface>, <port>, <state-lifetime>, <source-state-lifetime>, <interest-state-lifetime>, <route-state-lifetime>, <context-name-state-lifetime>, <source-leave-backoff>, <otidmap>, <topicmap>, <routemap>, <namemap>, <clientactor>, <application-id>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    ...
  </srs>
  ...
</um-srs>
```

13.1.24 SRS Element "<application-id>"

Adds an "Application ID" for SRS monitoring statistics. Used with monitoring format "pb" (proto buff).

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Default Value:** (none)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <application-id>SRS1</application-id>
    ...
  </srs>
  ...
</um-srs>
```

13.1.25 SRS Element "<clientactor>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Children:** <request-stream-max-msg-count>, <record-queue-service-interval>, <batch-frame-max-record-count>, <batch-frame-max-datagram-size>

13.1.26 SRS Element "<batch-frame-max-datagram-size>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

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

13.1.27 SRS Element "<batch-frame-max-record-count>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

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

13.1.28 SRS Element "<record-queue-service-interval>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

Valid range: 1 .. 1000

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

13.1.29 SRS Element "<request-stream-max-msg-count>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

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

13.1.30 SRS Element "<namemap>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Children:** <shards>

13.1.31 SRS Element "<shards>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <otidmap>, <topicmap>, <routemap>, <namemap>

13.1.32 SRS Element "<routemap>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Children:** <shards>

13.1.33 SRS Element "<topicmap>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Children:** <shards>

13.1.34 SRS Element "<otidmap>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Children:** <shards>

13.1.35 SRS Element "<source-leave-backoff>"

Set how long an SRS delays before informing receivers about certain state changes in sources. There are conditions related to sources being deleted or timing out which can cause a receiver to "flap" - repeating cycling between connect (BOS) and disconnect (EOS). These conditions are usually related connectivity problems in the network. This element can eliminate, or at least slow down the flapping.

Valid range: 0 .. 60000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Default Value:** 500 (half sec)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <source-leave-backoff>1000</source-leave-backoff>
    ...
  </srs>
  ...
</um-srs>
```

13.1.36 SRS Element "<context-name-state-lifetime>"

Sets the value (in seconds) of a Store's context name information lifetime.

If a Store loses connection with SRS, the Context Name Information Record (CNIR) will be remembered by the SRS for a limited amount of time: the *context name state lifetime*. After that time expires, the SRS deletes the CNIR associated with that lost endpoint connection.

Zero is a special value which disables the timing of context name information of disconnected Stores. I.e. with zero, the CNIR of a lost Store is never deleted. This is generally not recommended as it can lead to unlimited memory growth in both the SRS and client contexts.

Note that SRS Element "<state-lifetime>" has no effect on this option.

See SRS State Lifetime for more information.

Valid range: 0 .. 20736000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Default Value:** 86400 (24 hours)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <context-name-state-lifetime>120</context-name-state-lifetime>
    ...
  </srs>
  ...
</um-srs>
```

13.1.37 SRS Element "<route-state-lifetime>"

Sets the value (in seconds) of a DRO's routing information lifetime.

If a DRO endpoint loses connection with SRS, the Domain Information Record (DIR) and the Route Information Record (RTIR) will be remembered by the SRS for a limited amount of time: the *route state lifetime*. After that time expires, the SRS deletes the DIR and RTIR associated with that lost endpoint connection.

Zero is a special value which disables the timing of routing information of disconnected DROs. I.e. with zero, the DIR and RTIR of a lost DRO are never deleted. This is generally not recommended as it can lead to unlimited memory growth in both the SRS and client contexts.

Note that SRS Element "<state-lifetime>" has no effect on this option.

See SRS State Lifetime for more information.

Valid range: 0 .. 20736000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Default Value:** 30 (seconds)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <route-state-lifetime>120</route-state-lifetime>
    ...
  </srs>
  ...
</um-srs>
```

13.1.38 SRS Element "<interest-state-lifetime>"

Sets the value (in seconds) of the client interest state lifetime.

If a client context loses connection with SRS, the topic interest of that context will be remembered by the SRS for a limited amount of time: the *interest state lifetime*. If the context does not re-connect within that time, the SRS deletes all of the topic interest owned by that lost context.

Zero is a special value which disables the timing of interest of disconnected contexts. I.e. with zero, the interest of a lost context is never deleted. This is generally not recommended as it can lead to unlimited memory growth in both the SRS and client contexts.

If this element is not supplied, the interest state lifetime defaults to the current value for SRS Element "<state-lifetime>".

See SRS State Lifetime for more information.

Valid range: 0 .. 20736000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Default Value:** Current value for SRS Element "<state-lifetime>"

Example: (SRS deletes a lost context's interest after 120 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <interest-state-lifetime>120</interest-state-lifetime>
    ...
  </srs>
  ...
</um-srs>
```

13.1.39 SRS Element "<source-state-lifetime>"

Sets the value (in seconds) of the client source state lifetime.

If a client context loses connection with SRS, the sources contained by that context will be remembered by the SRS for a limited amount of time: the *source state lifetime*. If the context does not re-connect within that time, the SRS deletes all of the sources owned by that lost context. Those deletions will be shared with all connected client contexts.

Zero is a special value which disables the timing of sources of disconnected contexts. I.e. with zero, the sources from a lost context are never deleted. This is generally not recommended as it can lead to unlimited memory growth in both the SRS and client contexts.

If this element is not supplied, the source state lifetime defaults to the current value for SRS Element "<state-lifetime>"

See SRS State Lifetime for more information.

Valid range: 0 .. 20736000

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <srs>
- **Default Value:** Current value for SRS Element "<state-lifetime>"

Example: (SRS deletes a lost context's sources after 120 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <source-state-lifetime>120</source-state-lifetime>
    ...
  </srs>
  ...
</um-srs>
```

13.1.40 SRS Element "<state-lifetime>"

Sets the default (in seconds) of the client SRS Element "<source-state-lifetime>" and SRS Element "<interest-state-lifetime>". Note that it does **not** affect SRS Element "<route-state-lifetime>" or SRS Element "<context-name-state-lifetime>".

See SRS State Lifetime for more information.

Valid range: 0 .. 20736000

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

- **Default Value:** 86400 (24 hours)

Example: (SRS deletes a lost context's sources and interest after 120 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <state-lifetime>120</state-lifetime>
    ...
  </srs>
  ...
</um-srs>
```

13.1.41 SRS Element "<daemon>"

Contains elements which define logging behavior and sets a file name for the service's Process ID.

See child elements for details.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <um-srs>
- **Children:** <log>, <pid-file>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    ...
  </daemon>
  ...
</um-srs>
```

13.1.42 SRS Element "<pid-file>"

Supplies the desired name of file in which the SRS writes its Process ID (PID).

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

Example: (SRS writes process ID to "srs_pid.txt" file)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <pid-file>srs_pid.txt</pid-file>
    ...
  </daemon>
  ...
</um-srs>
```

13.1.43 SRS Element "<log>"

Configures SRS logging behavior. The value contained within the <log>...</log> is a file name, but is only used if the "type" attribute is set to "file".

When the type attribute is set to "file", the SRS supports "rolling" the log file, which consists a series of files over time so that no one file grows too large.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **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 the log file each hour.	**"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.	positiveInteger	**"10"** (10,485,760 bytes)
max-history	Number of rolled log files at which the oldest file is deleted when the current log file is rolled.	positiveInteger	**"10"**
total-size-cap	Total disk space consumed (in MB, i.e. 2**20, or 1,048,576) by rolled log files at which the oldest file is deleted to make room for the next log roll.	positiveInteger	**"1000"** (1,048,576,000 bytes)
compression	Enables compression for rolled log files.	**"none"** - Do not compress log files. **"zip"** - Compress log files using "zip" format. **"gzip"** - Compress log files using "gzip" format.	**"none"**

Example 1: (write log messages to standard out)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <log type="console"/>
    ...
  </daemon>
  ...
</um-srs>
```

Example 2: (write log messages to "srs.log" file)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <log type="file" frequency="daily">srs.log</log>
    ...
  </daemon>
  ...
</um-srs>
```

13.2 SRS XSD file

The XSD file is used to validate the user's configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="
      http://www.w3.org/2001/XMLSchema">
  <xs:element name="um-srs" type="um-srsType"/>

  <!-- Custom types and restrictions -->
  <xs:simpleType name="nonEmptyString">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="logTypeEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="file"/>
      <xs:enumeration value="console"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="logFrequencyEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="disable"/>
      <xs:enumeration value="daily"/>
      <xs:enumeration value="hourly"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="compressionEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="none"/>
      <xs:enumeration value="zip"/>
      <xs:enumeration value="gzip"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="portInteger">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="65535"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="booleanEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="true"/>
      <xs:enumeration value="false"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Acceptable values for publishingIntervalLong type are 0 or >= 200 -->
  <xs:simpleType name="publishingIntervalLong">
    <xs:union>
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:pattern value="0"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:minInclusive value="200"/>
          <xs:maxInclusive value="7776000000"/> <!-- 90 days in milliseconds -->
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>

  <!-- Acceptable values for stateLifetimeInteger type are 0, ..., 20736000 -->
  <xs:simpleType name="stateLifetimeInteger">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="20736000"/> <!-- 240 days in seconds -->
    </xs:restriction>
  </xs:simpleType>

  <!-- Acceptable values for sourceLeaveBackoffInteger type are 0, ..., 60000 -->
  <xs:simpleType name="sourceLeaveBackoffInteger">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="60000"/> <!-- 1 minute in milliseconds -->
    </xs:restriction>
  </xs:simpleType>

  <!-- Acceptable values for recordQueueServiceIntervalInteger type are 1, ..., 1000 -->
  <xs:simpleType name="recordQueueServiceIntervalInteger">
    <xs:restriction base="xs:positiveInteger">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="1000"/> <!-- 1 second in milliseconds -->
    </xs:restriction>
  </xs:simpleType>

  <!-- Acceptable values for pingIntervalInteger type are 100, ..., 3600000 -->
  <xs:simpleType name="pingIntervalInteger">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:minInclusive value="100"/>
      <xs:maxInclusive value="3600000"/> <!-- 1 hour in milliseconds -->
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="monitorFormatEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="json"/>
      <xs:enumeration value="pb"/>
    </xs:restriction>
  </xs:simpleType>


  <xs:complexType name="logType" >
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="logTypeEnumeration" name="type" use="required"/>
        <xs:attribute type="logFrequencyEnumeration" name="frequency"/>
        <xs:attribute type="xs:positiveInteger" name="size"/>
        <xs:attribute type="xs:positiveInteger" name="max-history"/>
        <xs:attribute type="xs:positiveInteger" name="total-size-cap"/>
        <xs:attribute type="compressionEnumeration" name="compression"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="otidmapType">
    <xs:all>
      <xs:element type="xs:positiveInteger" name="shards" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="topicmapType">
    <xs:all>
      <xs:element type="xs:positiveInteger" name="shards" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="routemapType">
  <xs:all>
    <xs:element type="xs:positiveInteger" name="shards" minOccurs="0" maxOccurs="1"/>
  </xs:all>
  </xs:complexType>
  <xs:complexType name="namemapType">
    <xs:all>
      <xs:element type="xs:positiveInteger" name="shards" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="clientActorType">
    <xs:all>
      <xs:element type="xs:positiveInteger" name="request-stream-max-msg-count" minOccurs="0" maxOccurs="1"
      />
      <xs:element type="recordQueueServiceIntervalInteger" name="record-queue-service-interval" minOccurs="
      0" maxOccurs="1"/>
      <xs:element type="xs:positiveInteger" name="batch-frame-max-record-count" minOccurs="0" maxOccurs="1"
      />
      <xs:element type="xs:positiveInteger" name="batch-frame-max-datagram-size" minOccurs="0" maxOccurs="1
      "/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="daemonType" >
    <xs:all>
      <xs:element type="logType" name="log" minOccurs="0" maxOccurs="1"/>
      <xs:element type="nonEmptyString" name="pid-file" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
```

```
  <xs:complexType name="debugMonitorType">
    <xs:all>
      <xs:element type="nonEmptyString" name="interface" minOccurs="0" maxOccurs="1"/>
      <xs:element type="portInteger" name="port" minOccurs="0" maxOccurs="1"/>
      <xs:element type="pingIntervalInteger" name="ping-interval" minOccurs="0" maxOccurs="1"/>
      <xs:element type="booleanEnumeration" name="enabled" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="lbmOptionType" >
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="nonEmptyString" name="scope" use="required"/>
        <xs:attribute type="nonEmptyString" name="name" use="required"/>
        <xs:attribute type="nonEmptyString" name="value" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="lbmAttributesType">
    <xs:sequence>
      <xs:element type="lbmOptionType" name="option" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="daemonMonitorType" mixed="true">
    <xs:all>
      <xs:element type="pingIntervalInteger" name="ping-interval" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalType" name="publishing-interval" minOccurs="0" maxOccurs="1"/>
      <xs:element type="lbmAttributesType" name="lbm-attributes" minOccurs="0" maxOccurs="1"/>
      <xs:element type="allowType" name="publish-connection-events" minOccurs="0" maxOccurs="1"/>
      <xs:element type="allowType" name="remote-snapshot-request" minOccurs="0" maxOccurs="1"/>
      <xs:element type="allowType" name="remote-config-changes-request" minOccurs="0" maxOccurs="1"/>
      <xs:element type="monitorFormatEnumeration" name="monitor-format" minOccurs="0" maxOccurs="1"/>
    </xs:all>
    <xs:attribute type="nonEmptyString" name="topic"/>
  </xs:complexType>
  <xs:complexType name="publishingIntervalType">
    <xs:all>
      <xs:element type="publishingIntervalLong" name="default" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="srs-stats" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="um-client-stats" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="connection-events" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="srs-error-stats" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="um-client-error-stats" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="config-opts" minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="internal-config-opts" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="allowType" >
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="booleanEnumeration" name="allow" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="um-srsType">
    <xs:all>
      <xs:element type="daemonType" name="daemon" minOccurs="0" maxOccurs="1"/>
      <xs:element type="srsType" name="srs" minOccurs="0" maxOccurs="1"/>
      <xs:element type="debugMonitorType" name="debug-monitor" minOccurs="0" maxOccurs="1"/>
      <xs:element type="daemonMonitorType" name="daemon-monitor" minOccurs="0" maxOccurs="1"/>
    </xs:all>
    <xs:attribute type="nonEmptyString" name="version"/>
  </xs:complexType>
  <xs:complexType name="srsType">
    <xs:all>
      <xs:element type="nonEmptyString" name="interface" minOccurs="0" maxOccurs="1"/>
      <xs:element type="portInteger" name="port" minOccurs="0" maxOccurs="1"/>
      <xs:element type="stateLifetimeInteger" name="state-lifetime" minOccurs="0" maxOccurs="1"/>
      <xs:element type="stateLifetimeInteger" name="source-state-lifetime" minOccurs="0" maxOccurs="1"/>
      <xs:element type="stateLifetimeInteger" name="interest-state-lifetime" minOccurs="0" maxOccurs="1"/>
      <xs:element type="stateLifetimeInteger" name="route-state-lifetime" minOccurs="0" maxOccurs="1"/>
      <xs:element type="stateLifetimeInteger" name="context-name-state-lifetime" minOccurs="0" maxOccurs="1
      "/>
      <xs:element type="sourceLeaveBackoffInteger" name="source-leave-backoff" minOccurs="0" maxOccurs="1"/
      >
      <xs:element type="otidmapType" name="otidmap" minOccurs="0" maxOccurs="1"/>
      <xs:element type="topicmapType" name="topicmap" minOccurs="0" maxOccurs="1"/>
      <xs:element type="routemapType" name="routemap" minOccurs="0" maxOccurs="1"/>
      <xs:element type="namemapType" name="namemap" minOccurs="0" maxOccurs="1"/>
      <xs:element type="clientActorType" name="clientactor" minOccurs="0" maxOccurs="1"/>
      <xs:element type="nonEmptyString" name="application-id" minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
</xs:schema>
```

Chapter 14

SRS Daemon Statistics

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

The SRS Daemon Statistics are published in the form of JSON messages. These are ASCII text messages which represent internal SRS data structures containing statistical and configuration information.

The following sub-sections describe the content of the messages. Note that while the sample messages shown are "beautified" (whitespace inserted for readability), a receiver of these messages should make no assumption about the presence or absence of whitespace. Also, as it true generally with JSON, the order of the fields is not fixed and can vary.

The message types are:

- Message Type: SRS_STATS
- Message Type: SRS_ERROR_STATS
- Message Type: UM_CLIENT_STATS
- Message Type: UM_CLIENT_ERROR_STATS
- Message Type: CONNECTION_EVENTS
- Message Type: CONFIG_OPTS
- Message Type: INTERNAL_CONFIG_OPTS
- Request Type: REPORT_SRS_VERSION
- Request Type: REPORT_MONITOR_INFO
- Request Type: SET_PUBLISHING_INTERVAL

14.1 Message Type: SRS_STATS

Message type SRS_STATS contains information about the overall state of the SRS service.

EXAMPLE:

```
{
  "monitorInfoCategory": "SRS_STATS",
  "stats": [
    {
      "name": "clients.inactive.SIR.count",
      "value": 0
    },
    {
      "name": "clients.next.client.ID",
      "value": 17
    }
  ]
}
```

This example has two statistics. Be aware that a given message can have any number of statistic entries.

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "SRS_STATS".
stats	Array of sub-structures, one per statistic. The number and order of the contained statistics is not fixed.
. . stats[].name	Name of statistic (see below).
. . stats[].value	Value of statistic.

Meaning of each statistic:

Statistic	Description
clients.next.client.ID	Unique session ID that will be assigned to the next context to connect.
active.clients.count	Number of currently connected contexts.
clients.connects.count	Number of contexts that have connected since this SRS started.
clients.disconnects.count	Number of contexts that have disconnected since this SRS started.
clients.max.concurrent.connections.count	High water mark of simultaneous connections since the SRS service was started.
clients.active.SIR.count	Number of sources being maintained from connected contexts.
clients.active.RIR.count	Number of receivers being maintained from connected contexts.
clients.active.WIR.count	Number of wildcard receivers being maintained from connected contexts.
clients.active.DIR.count	Number of Domain Information Records (DIRs) being maintained from connected DRO endpoints.
clients.inactive.SIR.count	Number of sources being temporarily maintained from disconnected contexts. These get cleaned up after the state lifetime expires.
clients.inactive.RIR.count	Number of receivers being temporarily maintained from disconnected contexts. These get cleaned up after the state lifetime expires.

Statistic	Description
clients.inactive.WIR.count	Number of wildcard receivers being temporarily maintained from disconnected contexts. These get cleaned up after the state lifetime expires.
clients.inactive.DIR.count	Number of Domain Information Records (DIRs) being temporarily maintained from disconnected DRO endpoints. These get cleaned up after the state lifetime expires.
clients.expired.SIR.count	Number of times the SRS deleted source records due to the state lifetime being expired. This will happen when applications exit without deleting their sources, which is not recommended. If this number increases frequently, consider modifying your applications to clean up before exiting.
clients.expired.RIR.count	Number of times the SRS deleted receiver records due to the state lifetime being expired. This will happen when applications exit without deleting their receivers, which is not recommended. If this number increases frequently, consider modifying your applications to clean up before exiting.
clients.expired.WIR.count	Number of times the SRS deleted wildcard receiver records due to the state lifetime being expired. This will happen when applications exit without deleting their receivers, which is not recommended. If this number increases frequently, consider modifying your applications to clean up before exiting.
clients.expired.DIR.count	Number of times the SRS deleted Domain Information Records (DIRs) due to the state lifetime being expired. This will happen when DROs exit abnormally. Informatica support should be informed if this count increments frequently.
clients.expired.CNIR.count	Number of times the SRS deleted Context Name Information Records (CNIRs) due to the state lifetime being expired. This will happen when Stores exit abnormally. Informatica support should be informed if this count increments frequently.
clients.DR.inactive.SIR.count	If DROs are involved, it is normal for this count to increment. Otherwise, this should not increment, and Informatica support should be informed if it does. (A context disconnected while it has SIRs that were inactive.)
clients.SLR.no.OTID.match.count	If the system is otherwise behaving normally, increments in this count are most likely harmless. (A receiver sent a source leave record for a source that the SRS does not know about.)
clients.active.RTIR.count	Number of Route Information Records (RTIR) being maintained from all connected DRO endpoint contexts. Should be equal to the number of active DRO endpoints in this TRD.
clients.inactive.RTIR.count	Number of Route Information Records (RTIR) being temporarily maintained from disconnected DRO endpoint contexts.
clients.expired.RTIR.count	Number of inactive Route Information Records (RTIR) deleted due to expiration of their state lifetimes. This value is cumulative since the SRS was started.

Statistic	Description
clients.active.CNIR.count	Number of Context Name Information Records (CNIRs) being maintained from connected contexts. This is for Stores identified by their context names; see **Identifying Persistent Stores**.
clients.inactive.CNIR.count	Number of Context Name Information Records (CNIRs) being temporarily maintained from disconnected contexts. Currently, this is only for Stores identified by their context names; see **Identifying Persistent Stores**.
clients.duplicate.CNIR.count	Number of duplicate Context Name Information Records (CN↩IRs) received from all Store endpoints. While not a fatal condition, Informatica support should be informed if this is non-zero.

All of the above statistics are included in a snapshot. Only the changed statistics are included during a periodic update.

14.2 Message Type: SRS_ERROR_STATS

Message type SRS_ERROR_STATS contains counters for errors detected by the SRS service. These types of errors should not be happening in a properly configured network; contact Informatica Support if the counters are increasing frequently.

EXAMPLE:

```
{
  "monitorInfoCategory": "SRS_ERROR_STATS",
  "stats": [
    {
      "name": "clients.duplicate.SIR.count",
      "value": 0
    },
    {
      "name": "clients.invalid.SDR.no.OTID.match.count",
      "value": 17
    }
  ]
}
```

This example has two statistics. Be aware that a given message can have any number of statistic entries.

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "SRS_ERROR_STATS".
stats	Array of sub-structures, one per statistic. The number and order of the contained statistics is not fixed.
. . stats[].name	Name of statistic (see below).
. . stats[].value	Value of statistic.

Meaning of each statistic:

Statistic	Description
clients.duplicate.SIR.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (Number of times that the SRS is informed about a source when it didn't need to be informed; i.e. the SRS already knew about it.)
clients.duplicate.RIR.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (Number of times that the SRS is informed about a receiver when it didn't need to be informed; i.e. the SRS already knew about it.)
clients.duplicate.WIR.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (Number of times that the SRS is informed about a wildcard receiver when it didn't need to be informed; i.e. the SRS already knew about it.)
clients.invalid.SDR.no.topic.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A context deleted a source that the SRS does not know about, which should never happen.)
clients.invalid.SDR.no.OTID.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A context deleted a source that the SRS does not know about, which should never happen.)
clients.invalid.SDR.no.transport.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A context deleted a source that the SRS does not know about, which should never happen.)
clients.invalid.DR.no.topic.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A source that the SRS does not know about disconnected, which should never happen.)
clients.invalid.DR.no.OTID.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A context deleted a source that the SRS does not know about, which should never happen.)
clients.invalid.DR.no.transport.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A context deleted a source that the SRS does not know about, which should never happen.)
clients.invalid.SLR.no.topic.match.count	While not a fatal condition, Informatica support should be informed if this count increments frequently. (A receiver notified source leave for a source that the SRS does not know about, which should never happen.)
clients.invalid.DR.inactive.SIR.count	This counter should never be greater than zero. While not a fatal condition, Informatica support should be informed if this count increments frequently. (A context disconnected while it has SIRs that were inactive, which should never happen.)

Statistic	Description
clients.duplicate.DIR.count	Number of duplicate Domain Information Records (DIRs) the SRS generated. This should never happen. While not a fatal condition, Informatica support should be informed if this is non-zero.
clients.mismatched.DIR.count	A non-zero count indicates that at least two DRO endpoints have reported different domain IDs, which is illegal. All DRO endpoints servicing the same TRD must be configured for the same domain ID value. If all DRO endpoints being services by this SRS are configured for the same <**domain-id**> value, contact Informatica Support.
clients.duplicate.RTIR.count	Number of duplicate Route Information Records (RTIRs) received from all DRO endpoints. While not a fatal condition, Informatica support should be informed if this is non-zero. See also statistic "client.duplicate.RTIR.received.count" in Message Type: UM_CLIENT_ERROR_STATS.
clients.invalid.CNIR.name.mismatch.count	Number of Context Name Information Records (CNIRs) received from a context that are different from the initial CNIR. This should never happen. While not a fatal condition, Informatica support should be informed if this is non-zero.

14.3 Message Type: UM_CLIENT_STATS

Message type UM_CLIENT_STATS contains information related to an individual connected context. Multiple instances of this message can be sent, one per connected context.

EXAMPLE:

```
{
  "monitorInfoCategory": "UM_CLIENT_STATS",
  "srsRegistrationInfo": {
    "ip": "10.29.3.43",
    "port": "60681",
    "sessionId": "0x9739b88f"
  },
  "stats": [
    {
      "name": "client.SIR.received.count",
      "value": 1
    },
    {
      "name": "client.SDR.received.count",
      "value": 0
    }
  ]
}
```

This example has two statistics. Be aware that a given message can have any number of statistic entries.

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "UM_CLIENT_STATS".
srsRegistrationInfo	Structure containing identifying information about the connected context.
. . srsRegistrationInfo.ip	IP address of the context.
. . srsRegistrationInfo.port	TCP "source port" the context used locally for its SRS connection. See **resolver_service (context)**.
. . srsRegistrationInfo.sessionId	The unique identifier assigned by the SRS to this connection.
stats	Array of sub-structures, one per statistic. The number and order of the contained statistics is not fixed.
. . stats[].name	Name of statistic (see below).
. . stats[].value	Value of statistic.

Meaning of each statistic:

Statistic	Description
record.queue.depth	Snapshot of the context's SRS worker thread's work queue. I.e. number of Topic Resolution updates that need to be sent to the context. This number should normally be zero. If it remains above zero for significant time `contact Informatica Support`.
client.SIR.sent.count	Number of sources that the SRS has informed the context of.
client.SER.sent.count	Number of source deletions that the SRS has informed the context of. Either the application explicitly deleted a source, or the context abnormally disconnected and the state lifetime expired.
client.RIR.sent.count	Number of receivers that the SRS has informed the context of.
client.RER.sent.count	Number of receiver deletions that the SRS has informed the context of. Either the application explicitly deleted a receiver, or the context abnormally disconnected and the state lifetime expired.
client.WIR.sent.count	Number of wildcard receivers that the SRS has informed the context of.
client.WER.sent.count	Number of wildcard receiver deletions that the SRS has informed the context of. Either the application explicitly deleted a receiver, or the context abnormally disconnected and the state lifetime expired.
client.SIR.received.count	Number of created sources that the context has informed the S↩ RS. This number is not necessarily the current number of sources; i.e. the counter does not decrease as sources are deleted.
client.SDR.received.count	Number of sources deleted that the context has informed the SRS.
client.RIR.received.count	Number of created receivers (one per topic even if more than one receivers on a topic) that the context has informed the SRS. This number is not necessarily the current number of receivers; i.e. the counter does not decrease as receivers are deleted.

Statistic	Description
client.WIR.received.count	Number of created wildcard receivers (one per pattern even if more than one receivers on a pattern) that the context has informed the SRS. This number is not necessarily the current number of wildcard receivers; i.e. the counter does not decrease as wildcard receivers are deleted.
client.RDR.received.count	Number of deleted receivers (one per topic even if more than one receivers on a topic) that the context has informed the SRS. This number is not necessarily the current number of receivers; i.e. the counter does not decrease as receivers are deleted.
client.WDR.received.count	Number of deleted wildcard receivers (one per pattern even if more than one receivers on a pattern) that the context has informed the SRS. This number is not necessarily the current number of wildcard receivers; i.e. the counter does not decrease as wildcard receivers are deleted.
client.SLR.received.count	Number of source leave records that the receiver context has informed the SRS.
client.active.SIR.count	Number of sources that currently exist in the context.
client.max.concurrent.SIR.count	High water mark of simultaneous sources managed since the SRS service started.
client.DIR.sent.count	Number of Domain Information Records (DIRs) that the SRS has sent to the context.
client.RTIR.sent.count	Number of Route Information Records (RTIRs) the SRS has sent to the context.
client.RTIR.received.count	Number of Route Information Records (RTIRs) the context has sent to the SRS.
client.RTER.sent.count	Number of Route End Records (RTERs) the SRS has sent to the context. This is incremented if a DRO disconnects from an SRS and that DRO's Route Information Record state lifetime expires.
client.CNIR.sent.count	Number of Context Name Information Records (CNIRs) sent to the context.
client.CNIR.received.count	Number of Context Name Information Records (CNIRs) received from the context. This should normally be 1 or 2.
client.CNQR.received.count	Number of Context Name Query Records (CNQRs) received from the context. This should normally be 1.
client.unexpected.CNER.received.count	Number of Context Name End Records (CNERs) received from the context. This should normally be 0, but if the client is UM version 6.14, this counter can increase.

14.4 Message Type: UM_CLIENT_ERROR_STATS

Message type UM_CLIENT_ERROR_STATS contains error counters related to an individual connected context. Multiple instances of this message can be sent, one per connected context. These types of errors should not be happening in a properly configured network; contact Informatica Support if the counters are increasing frequently.

EXAMPLE:

```
{
  "monitorInfoCategory": "UM_CLIENT_ERROR_STATS",
  "srsRegistrationInfo": {
    "ip": "10.29.3.43",
    "port": "60681",
    "sessionId": "0x9739b88f"
  },
  "stats": [
    {
      "name": "client.invalid.SRS.message.received.count",
      "value": 1
    },
    {
      "name": "client.invalid.SDR.received.count",
      "value": 0
    }
  ]
}
```

This example has two statistics. Be aware that a given message can have any number of statistic entries.

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "UM_CLIENT_ERROR_STATS".
srsRegistrationInfo	Structure containing identifying information about the connected context.
. . srsRegistrationInfo.ip	IP address of the context.
. . srsRegistrationInfo.port	TCP "source port" the context used locally for its SRS connection. See **resolver_service (context)**.
. . srsRegistrationInfo.sessionId	The unique identifier assigned by the SRS to this connection.
stats	Array of sub-structures, one per statistic. The number and order of the contained statistics is not fixed.
. . stats[].name	Name of statistic (see below).
. . stats[].value	Value of statistic.

Meaning of each statistic:

Statistic	Description
client.duplicate.RIR.received.count	Number of created duplicate receivers that the context has informed the SRS.
client.duplicate.WIR.received.count	Number of created duplicate wildcard receivers that the context has informed the SRS.
client.invalid.SRS.message.received.count	This counter should never be greater than zero. While not a fatal condition, Informatica support should be informed if this count increments frequently. (Number of messages received from this context that could not be processed.)

Statistic	Description
client.invalid.SDR.received.count	This counter should never be greater than zero. While not a fatal condition, Informatica support should be informed if this count increments frequently. (Number of source delete messages received from this context which could not be processed correctly.)
client.duplicate.RTIR.received.count	Number of duplicate Route Information Records (RTIRs) received from this DRO endpoint. This should never happen. While not a fatal condition, Informatica support should be informed if this is non-zero. See also statistic "clients.duplicate.RTIR.count" in Message Type: SRS_ERROR_STATS.
client.unexpected.SRS.message.sent.count	Number of unexpected messages sent to the context. This should never happen. While not a fatal condition, Informatica support should be informed if this is non-zero.
client.duplicate.CNIR.received.count	Number of duplicate Context Name Information Records (C↩NIRs) received from this context. This should never happen. While not a fatal condition, Informatica support should be informed if this is non-zero.
client.duplicate.CNQR.received.count	Number of duplicate Context Name Query Records (CNQRs) received from this context. This should never happen. While not a fatal condition, Informatica support should be informed if this is non-zero.

14.5 Message Type: CONNECTION_EVENTS

14.5.1 Message Subtype: UM_CLIENT_CONNECT

Message type CONNECTION_EVENTS, sub-type UM_CLIENT_CONNECT, logs a single connect of a context to the SRS.

```
{
  "monitorInfoCategory": "CONNECTION_EVENTS",
  "srsRegistrationInfo": {
    "ip": "10.29.3.42",
    "port": "41873",
    "sessionId": "0xaecbff98"
  },
  "connectionEventType": "UM_CLIENT_CONNECT",
  "events": [
    {
      "connectionEventType": "UM_CLIENT_CONNECT",
      "connectionEventTime": "Thu Jan 24 00:28:26 CET 2019",
    }
  ]
}
```

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "CONNECTION_EVENTS".
srsRegistrationInfo	Structure containing identifying information about the connected context.
. . srsRegistrationInfo.ip	IP address of the context.
. . srsRegistrationInfo.port	TCP "source port" the context used locally for its SRS connection. See **resolver_service (context)**.
. . srsRegistrationInfo.sessionId	The unique identifier assigned by the SRS to this connection.
connectionEventType	type of event contained in the `events` sub-structure. Set to "UM_CLIENT_CONNECT".
events	Technically `events` is constructed as an array of sub-structures, however each CONNECTION_EVENTS message contains exactly one event log.
. . events[].connectionEventType	type of event contained in the `events` sub-structure. Set to "UM_CLIENT_CONNECT".
. . events[].connectionEventTime	ASCII time/date stamp of event.

14.5.2 Message Subtype: UM_CLIENT_DISCONNECT

Message type CONNECTION_EVENTS, sub-type UM_CLIENT_DISCONNECT, logs a single disconnect of a context to the SRS.

```
{
  "monitorInfoCategory": "CONNECTION_EVENTS",
  "srsRegistrationInfo": {
    "ip": "10.29.3.42",
    "port": "35350",
    "sessionId": "0x0388cf20"
  },
  "connectionEventType": "UM_CLIENT_DISCONNECT",
  "events": [
    {
      "srsRegistrationInfo": {
        "ip": "10.29.3.42",
        "port": "35350",
        "sessionId": "0x0388cf20"
      },
      "connectionEventType": "UM_CLIENT_DISCONNECT",
      "connectionEventTime": "Wed Jan 30 23:50:30 CET 2019",
    }
  ]
}
```

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "CONNECTION_EVENTS".
srsRegistrationInfo	Structure containing identifying information about the connected context.

Field	Description
. . srsRegistrationInfo.ip	IP address of the context.
. . srsRegistrationInfo.port	TCP "source port" the context used locally for its SRS connection. See **resolver_service (context)**.
. . srsRegistrationInfo.sessionId	The unique identifier assigned by the SRS to this connection.
connectionEventType	type of event contained in the `events` sub-structure. Set to "UM_CLIENT_DISCONNECT".
events	Technically `events` is constructed as an array of sub-structures, however each CONNECTION_EVENTS message contains exactly one event log.
. . events[].srsRegistrationInfo	Structure containing identifying information about the connected context.
. . . . events[].srsRegistrationInfo.ip	IP address of the context.
. . . . events[].srsRegistrationInfo.port	TCP "source port" the context used locally for its SRS connection. See **resolver_service (context)**.
. . . . event[].srsRegistrationInfo.sessionId	The unique identifier assigned by the SRS to this connection.
. . events[].connectionEventType	type of event contained in the `events` sub-structure. Set to "UM_CLIENT_DISCONNECT".
. . events[].connectionEventTime	ASCII time/date stamp of event.

14.5.3 Message Subtypes: SIR and SDR

Message type CONNECTION_EVENTS, sub-types SIR and SDR, log a single context source creation or deletion message to the SRS.

```
{
  "monitorInfoCategory": "CONNECTION_EVENTS",
  "srsRegistrationInfo": {
    "ip": "10.29.3.43",
    "port": "60809",
    "sessionId": "0x1c668bad"
  },
  "connectionEventType": "SIR" or "SDR",
  "events": [
    {
      "topic": "srs_topic",
      "source": "LBTRM:10.29.3.43:24000:47b87920:225.11.28.85:14400",
      "connectionEventType": "SIR" or "SDR",
      "connectionEventTime": "Fri Feb 1 02:16:06 CET 2019",
    }
  ]
}
```

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "CONNECTION_EVENTS".
srsRegistrationInfo	Structure containing identifying information about the connected context.
. . srsRegistrationInfo.ip	IP address of the context.
. . srsRegistrationInfo.port	TCP "source port" the context used locally for its SRS connection. See **resolver_service (context)**.
. . srsRegistrationInfo.sessionId	The unique identifier assigned by the SRS to this connection.
connectionEventType	type of event contained in the `events` sub-structure. Set to "SIR" or "SDR".
events	Technically `events` is constructed as an array of sub-structures, however each CONNECTION_EVENTS message contains exactly one event log.
. . events[].connectionEventType	type of event contained in the `events` sub-structure. Set to "SIR" or "SDR".
. . events[].connectionEventTime	ASCII time/date stamp of event.

14.6 Message Type: CONFIG_OPTS

Message type CONFIG_OPTS contains SRS configuration information.

EXAMPLE:

```
{
  "monitorInfoCategory": "CONFIG_OPTS",
  "configOptions": [
    {
      "name": "um-srs.daemon-monitor.lbm-attributes.context.context_name",
      "value": "statsLbmContext",
    },
    {
      "name": "um-srs.daemon-monitor.lbm-attributes.context.default_interface",
      "value": "192.168.0.0/24",
    }
  ]
}
```

This example has two options. Be aware that a given message can have any number of option entries.

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "CONFIG_OPTS".
configOptions	Array of sub-structures, one per configuration option. The number and order of the contained options is not fixed.
. . configOptions[].name	Name of option (see below).
. . configOptions[].value	Value of option.

Meaning of each option:

Option Name	Description
um-srs.version	Value for '**version**' attribute to SRS configuration element <um-srs>.
um-srs.daemon.log	Value for the SRS configuration element <log>.
um-srs.daemon.log.type	Value for '**type**' attribute to SRS configuration element <log>.
um-srs.daemon.log.frequency	Value for '**frequency**' attribute to SRS configuration element <log>.
um-srs.daemon.log.size	Value for '**size**' attribute to SRS configuration element <log>.
um-srs.daemon.log.max-history	Value for '**max-history**' attribute to SRS configuration element <log>.
um-srs.daemon.log.total-size-cap	Value for '**total-size-cap**' attribute to SRS configuration element <log>.
um-srs.daemon.log.compression	Value for '**compression**' attribute to SRS configuration element <log>.
um-srs.daemon.pid-file	Value for the SRS configuration element <request-stream-max-msg-count>.
um-srs.srs.interface	Value for the SRS configuration element <interface> inside <srs>.
um-srs.srs.port	Value for the SRS configuration element <port> inside <srs>.
um-srs.srs.state-lifetime	Value for the SRS configuration element <state-lifetime>.
um-srs.srs.source-state-lifetime	Value for the SRS configuration element <source-state-lifetime>.
um-srs.srs.interest-state-lifetime	Value for the SRS configuration element <interest-state-lifetime>.
um-srs.srs.route-state-lifetime	Value for the SRS configuration element <context-name-state-lifetime>.
um-srs.srs.context-name-state-lifetime	Value for the SRS configuration element <context-name-state-lifetime>.
um-srs.srs.source-leave-backoff	Value for the SRS configuration element <source-leave-backoff>.
um-srs.srs.application-id	Value for the SRS configuration element <application-id>.
um-srs.srs.clientactor.request-stream-max-msg-count	Value for the SRS configuration element <request-stream-max-msg-count>.

Option Name	Description
um-srs.srs.clientactor.record-queue-service-interval	Value for the SRS configuration element <record-queue-service-interval>.
um-srs.srs.clientactor.batch-frame-max-record-count	Value for the SRS configuration element <batch-frame-max-record-count>.
um-srs.srs.clientactor.batch-frame-max-datagram-size	Value for the SRS configuration element <batch-frame-max-datagram-size>.
um-srs.debug-monitor.interface	Value for the SRS configuration element <interface> inside <debug-monitor>.
um-srs.debug-monitor.port	Value for the SRS configuration element <port> inside <debug-monitor>.
um-srs.debug-monitor.enabled	Value for the SRS configuration element <enabled>.
um-srs.debug-monitor.ping-interval	Value for the SRS configuration element <ping-interval> for debug-monitor.
um-srs.daemon-monitor.ping-interval	Value for the SRS configuration element <ping-interval> for daemon-monitor.
um-srs.daemon-monitor.topic	Value for '**topic**' attribute to SRS configuration element <daemon-monitor>.
um-srs.daemon-monitor.publishing-interval.↩ default	Value for the SRS configuration element <default>.
um-srs.daemon-monitor.publishing-interval.srs-stats	Value for the SRS configuration element <srs-stats>.
um-srs.daemon-monitor.publishing-interval.um-client-stats	Value for the SRS configuration element <um-client-stats>.
um-srs.daemon-monitor.publishing-interval.↩ connection-events	Value for the SRS configuration element <connection-events>.
um-srs.daemon-monitor.publishing-interval.srs-error-stats	Value for the SRS configuration element <srs-error-stats>.
um-srs.daemon-monitor.publishing-interval.um-client-error-stats	Value for the SRS configuration element <um-client-error-stats>.
um-srs.daemon-monitor.publishing-interval.↩ config-opts	Value for the SRS configuration element <config-opts>.
um-srs.daemon-monitor.publishing-interval.↩ internal-config-opts	Value for the SRS configuration element <internal-config-opts>.
um-srs.daemon-monitor.publish-connection-events.allow	Value for '**allow**' attribute to SRS configuration element <publish-connection-events>.
um-srs.daemon-monitor.remote-snapshot-request.allow	Value for '**allow**' attribute to SRS configuration element <remote-snapshot-request>.
um-srs.daemon-monitor.remote-config-changes-request.allow	Value for '**allow**' attribute to SRS configuration element <remote-config-changes-request>.
um-srs.daemon-monitor.monitor-format	Value for the SRS configuration element <monitor-format>.

Option Name	Description
um-srs.daemon-monitor.lbm-attributes.lbm↩ConfigOptionScope.lbmConfigOptionName	A UM configuration option, as documented in **Configuration Overview**, where 'lbmConfigOptionScope' is the option scope ('context', 'source', etc) and 'lbm↩ConfigOptionName' is the option name.

14.7 Message Type: INTERNAL_CONFIG_OPTS

Message type INTERNAL_CONFIG_OPTS contains SRS internal configuration information. These options are not intended for application use.

EXAMPLE:

```
{
  "monitorInfoCategory": "INTERNAL_CONFIG_OPTS",
  "configOptions": [
    {
      "name": "um-srs.srs.otidmap.async-receiver-distribution",
      "value": "false",
    },
    {
      "name": "um-srs.srs.otidmap.shards",
      "value": "4",
    }
  ]
}
```

This example has two options. Be aware that a given message can have any number of option entries.

Overall structure of message:

Field	Description
monitorInfoCategory	Message type. Set to the string "INTERNAL_CONFIG_OPTS".
configOptions	Array of sub-structures, one per configuration option. The number and order of the contained options is not fixed.
. . configOptions[].name	Name of option (see below).
. . configOptions[].value	Value of option.

Meaning of each option:

Option Name	Description
um-srs.srs.otidmap.shards	Value for the SRS configuration element <shards> within element <otidmap>.
um-srs.srs.topicmap.shards	Value for the SRS configuration element <shards> within element <topicmap>.

Option Name	Description
um-srs.srs.routemap.shards	Value for the SRS configuration element <shards> within element <routemap>.
um-srs.srs.namemap.shards	Value for the SRS configuration element <shards> within element <namemap>.

14.8 Request Type: REPORT_SRS_VERSION

Request type REPORT_SRS_VERSION is sent by a monitoring application to determine the software version of the SRS.

```
{
  "commandMessageType": "REPORT_SRS_VERSION"
}
```

The SRS will send a response of the form:

```
SRS Version 6.16.1
```

14.9 Request Type: REPORT_MONITOR_INFO

Request type REPORT_MONITOR_INFO is sent by a monitoring application to initiate an immediate publishing of monitoring data.

The SRS will only process this request if the configuration contains <remote-snapshot-request allow="true"/>.

```
{
  "commandMessageType": "REPORT_MONITOR_INFO",
  "monitorInfoCategory": "SRS_STATS"
}
```

Where the "monitorInfoCategory" field is set to one of the following:

- SRS_STATS
- UM_CLIENT_STATS
- CONNECTION_EVENTS
- SRS_ERROR_STATS
- UM_CLIENT_ERROR_STATS
- CONFIG_OPTS
- INTERNAL_CONFIG_OPTS

The SRS will send a response of the form:

```
snap SRS_STATS - OK!
```

Note that "SRS_STATS" is replaced by the requested category.

14.10 Request Type: SET_PUBLISHING_INTERVAL

Request type SET_PUBLISHING_INTERVAL is sent by a monitoring application to Modify the publishing intervals for a running SRS. Note that SRS does not persist the new interval value; if the SRS is restarted, the value returns to the value configured via <daemon-monitor>.

The SRS will only process this request if the configuration contains <remote-config-changes-request allow="true"/>.

```
{
  "commandMessageType": "SET_PUBLISHING_INTERVAL",
  "monitorInfoCategory": "SRS_STATS",
  "publishingInterval": 60000
}
```

Where the "monitorInfoCategory" field is set to one of the following:

- SRS_STATS
- UM_CLIENT_STATS
- CONNECTION_EVENTS
- SRS_ERROR_STATS
- UM_CLIENT_ERROR_STATS
- CONFIG_OPTS
- INTERNAL_CONFIG_OPTS

The SRS will send a response of the form:

```
SRS_STATS 60000 - OK!
```

Note that "SRS_STATS" is replaced by the requested category.

Chapter 15

Man Pages for Lbmrd

Unicast UDP-based Topic Resolution services are provided by the Lbm Resolver Daemon (lbmrd).

For more information on UDP-based TR, see UDP-Based Topic Resolution Details. For more information on Topic Resolution general, see Topic Resolution Description.

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

- Lbmrd Man Page - Unix and Windows command-line interface.
- Lbmrds Man Page - Windows Service interface.

15.1 Lbmrd Man Page

Unix and Windows command-line interface.

```
UMResolver daemon
Usage: lbmrd [options] [config-file]
Available options:
  -a, --activity=IVL    interval between client activity checks (in
      milliseconds)(default 60000)
  -d, --dump-dtd        dump the configuration DTD to stdout and exit
  -h, --help            display this help and exit
  -i, --interface=ADDR  listen for unicast topic resolution messages on interface
      ADDR
                        ADDR accepts CIDR eg:10.0.0.0/8, Quoted device name
                            eg:"eth0", DNS Host name eg:host.mydomain.com/24.
  -L, --logfile=FILE    use FILE as the log file
  -p, --port=PORT       use UDP port PORT for topic resolution messages (default
      15380)
  -t, --ttl=TTL         use client time-to-live of TTL seconds (default 60)
  -r, --rcv-buf=SIZE    set the receive buffer to SIZE bytes.
  -s, --snd-buf=SIZE    set the send buffer to SIZE bytes.
  -v, --validate        validate config-file then exit
```

Description

The `lbmrd` command runs the Lbm Resolver Daemon. It can be run interactively from a shell or command prompt, or from a script or batch file. (For use as a Windows Service, see Lbmrds Man Page.)

The **"config-file"** parameter is optional. If supplied, it specifies the file path for the lbmrd's XML configuration file. If omitted, the lbmrd defaults all configuration details. See lbmrd Configuration File for configuration details.

The **"-i"** and **"-p"** options identify the network interface IP address and port that lbmrd opens to listen for unicast topic resolution traffic. The defaults are 0.0.0.0 and 15380, respectively. Note that 0.0.0.0 is not interpreted as INADDR_ANY, it is a directive for UM to choose the first interface it finds. See **Specifying Interfaces** for methods of specifying the interface. Alternatively, the LBMRD Element "<interface>" and LBMRD Element "<port>" can be used in the lbmrd Configuration File.

WARNING: It is strongly recommended to specify an interface when running lbmrd, either via the "-i" command-line option, or the <interface> XML tag. Otherwise, UM will select the first interface it finds, potentially 127.↩
0.0.1 (loopback), which is rarely a good choice. Note that CIDR notation can make it easier. For example, "10.0.0.0/8" will match any interface on the 10 network.

The **"-a"** and **"-t"** options interact to detect and remove "dead" clients, i.e., client applications that are in the lbmrd active client list, but have stopped sending topic resolution queries, advertisements, or keepalives, usually due to early termination or looping. These are described in detail below.

The **"-t"** option describes the length of time (in seconds), during which no messages have been received from a given client, that will cause that client to be marked "dead" and removed from the active client list. Ultra Messaging recommends a value at least 5 seconds longer than the longest network outage you wish to tolerate. Alternatively, the LBMRD Element "<ttl>" can be used in the lbmrd Configuration File.

Option **"-a"** describes a repeating time interval (in milliseconds), after which lbmrd checks for these "dead" clients. Ultra Messaging recommends a value not larger than "-t" ∗ 1000. Alternatively, the LBMRD Element "<activity>" can be used in the lbmrd Configuration File.

Note that even clients that send no topic resolution advertisements or queries will still send keepalive messages to lbmrd every 5 seconds. This value is hard-coded and not configurable.

The **"-s"** option sets the send socket buffer size in bytes. Alternatively, the LBMRD Element "<resolver_↩
unicast_send_socket_buffer>" can be used in the lbmrd Configuration File.

The **"-r"** option sets the receive socket buffer size in bytes. Alternatively, the LBMRD Element "<resolver_↩
unicast_receiver_socket_buffer>" can be used in the lbmrd Configuration File.

The **"-L"** option specifies the file path name for the lbmrd log file. Alternatively, the LBMRD Element "<log>" can be used in the lbmrd Configuration File.

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

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

Exit Status

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

15.2 Lbmrds Man Page

Windows Service interface.

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

Note that many operating parameters that are available in the `lbmrd` command are not available in the `lbmrds` command. For example, "-p", "-t", "-r", etc. If it is desired to set these parameters, the corresponding XML elements must be used in an lbmrd configuration file.

```
UMResolver service
Usage: lbmrds [options] [config-file]
Available options:
  -h, --help               display this help and exit
  -E, --env-var-file       update/set environment variable file
  -U, --unset-env-var-file  unset the environment variable file
  -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
  -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
  configfile               XML config file (if not present, looks in registry)
```

Description

The `lbmrds` 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 lbmrd Service. See **Configure the Windows Service**.
- Second, it provides Windows with the lbmrd Daemon executable to run as a Service.

The **"config-file"** parameter provides the file path for the lbmrd's XML configuration file. It is supplied in conjunction with the "-s config" option (see below). See lbmrd Configuration File for configuration details.

For **"-S install"** see **Install the Windows Service** (note that `lbmrds` uses upper-case "-S").

For **"-S remove"** see **Remove the Windows Service** (note that `lbmrds` uses upper-case "-S").

For **"-S config"**, **"-e"**, **"-E"**, and **"-U"**, see **Configure the Windows Service** (note that `lbmrds` uses upper-case "-S").

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

Exit Status

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

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.

Chapter 16

lbmrd Configuration File

The lbmrd configuration file must start with this line:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

After that, the '<**lbmrd**>' element contains the rest of the configuration.

The lbmrd configuration does not support the XInclude feature.

Note

> The configuration file must contain a '<**domains**>' element and a '<**transformations**>' element (and their contents), even if there is no NAT. See Dummy lbmrd Configuration File. The '<**daemon**>' element and its contents are optional.

16.1 lbmrd Configuration Elements

16.1.1 LBMRD Element "<lbmrd>"

Container element which holds the lbmrd configuration. Also defines the version of the configuration format used by the file.

- **Children:** <daemon>, <domains>, <transformations>

XML Attributes:

Attribute	Description	Valid Values	Default Value
version	Version number of user's configuration file.	"1.0" - Initial version	"1.0"

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  ...
</lbmrd>
```

16.1.2 LBMRD Element "<transformations>"

Container element for definitions of NAT translations applied to TIRs. Translations are used to help lbmrd know how to modify source advertisements when Network Address Translation (NAT) is being used.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <lbmrd>
- **Children:** <transform>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.3 LBMRD Element "<transform>"

Defines a set of transformation tuples. Each tuple applies to a TIR sent from a specific network domain (specified using the `source` attribute), and destined for a specific network domain (specified using the `destination` attribute). The `source` and `destination` attributes must specify network domain names as defined by the <domain> elements.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <transformations>
- **Children:** <rule>

XML Attributes:

Attribute	Description	Valid Values	Default Value
source	Name of source network domain, defined in <domain>.	IDREF	(no default; must be specified)
destination	Name of receiver network domain, defined in <domain>.	IDREF	(no default; must be specified)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.4 LBMRD Element "<rule>"

Container for a transformation rule which maps one address and port to another.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <transform>
- **Children:** <match>, <replace>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        ...
      </rule>
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.5 LBMRD Element "<replace>"

Defines the address and port which are to replace those matched in the TIR originating from a UM context within the `source` network (as specified by <transform>), and being delivered to contexts within the `destination` network.

- **Parent:** <rule>

XML Attributes:

Attribute	Description	Valid Values	Default Value
address	IP address within a TIR. Address must be specified only in dotted-decimal and refer to a specific host. For LBMRD Element "<match>", the IP address should be within the network specified by <transform> `source` attribute. For LBMRD Element "<replace>", the IP address should be within the network specified by <transform> `destination` attribute.	string	(no default; must be specified)
port	Port number to match or replace. To match any port, use value "*". To replace with same port as matched, use value "*".	string	"*"

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        <match address="10.1.1.50" port="*"/>
        <replace address="192.168.1.1" port="*"/>
        ...
      </rule>
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.6 LBMRD Element "<match>"

Defines the address and port to match within a TIR originating from a UM context within the `source` network (as specified by <transform>), and being delivered to contexts within the `destination` network.

- **Parent:** <rule>

XML Attributes:

Attribute	Description	Valid Values	Default Value
address	IP address within a TIR. Address must be specified only in dotted-decimal and refer to a specific host. For LBMRD Element "<match>", the IP address should be within the network specified by <transform> `source` attribute. For LBMRD Element "<replace>", the IP address should be within the network specified by <transform> `destination` attribute.	string	(no default; must be specified)
port	Port number to match or replace. To match any port, use value `"*"`. To replace with same port as matched, use value `"*"`.	string	`"*"`

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        <match address="10.1.1.50" port="*"/>
        <replace address="192.168.1.1" port="*"/>
        ...
      </rule>
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.7 LBMRD Element "<domains>"

Container element for definitions of network domains. Network domains are used to help lbmrd recognize networks and/or subnetworks which connect via Network Address Translation (NAT).

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <lbmrd>
- **Children:** <domain>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <domains>
    ...
  </domains>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.8 LBMRD Element "<domain>"

Defines a network domain. The domain must be given a unique name via the `name` attribute. This name is referenced in <transform> elements. The `<domain>` element contains one or more <network> elements.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <domains>
- **Children:** <network>

XML Attributes:

Attribute	Description	Valid Values	Default Value
name	Unique name assigned to the defined network.	ID	(no default; must be specified)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <domains>
    <domain name="Net-NYC">
      ...
    </domain>
    <domain name="Net-NJC">
      ...
    </domain>
    ...
  </domains>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.9 LBMRD Element "<network>"

Defines a single network specification which is to be considered part of the enclosing <domain> element. The network specification must contain either an IP address, or a network specification in `CIDR notation`. DNS host names are not supported in the lbmrd configuration file.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <domain>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <domains>
    <domain name="Net-NYC">
      <network>10.1.0.0/16</network>
      ...
    </domain>
    <domain name="Net-NJC">
      <network>192.168.1/24</network>
      ...
    </domain>
    ...
  </domains>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

16.1.10 LBMRD Element "<daemon>"

Container element for configuration related to the overall lbmrd process.

- **Cardinality** (number of times element can be supplied): 0 .. 1
- **Parent:** <lbmrd>
- **Children:** <activity>, <interface>, <port>, <ttl>, <log>, <resolver_unicast_receiver_socket_buffer>, <resolver_unicast_send_socket_buffer>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.11 LBMRD Element "<resolver_unicast_send_socket_buffer>"

Sets the send-side socket buffer size (in bytes).

- **Parent:** <daemon>
- **Default Value:** 1048576

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <resolver_unicast_send_socket_buffer>
      1048576
    </resolver_unicast_send_socket_buffer>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.12 LBMRD Element "<resolver_unicast_receiver_socket_buffer>"

Sets the receive-side socket buffer size (in bytes).

- **Parent:** <daemon>
- **Default Value:** 1048576

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <resolver_unicast_receiver_socket_buffer>
      1048576
    </resolver_unicast_receiver_socket_buffer>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.13 LBMRD Element "<log>"

Specifies the file name used for lbmrd logging.

- **Parent:** <daemon>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <log>lbmrd.log</log>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.14 LBMRD Element "<ttl>"

Interval (in milliseconds) between keep alive checks between the lbmrd and the UM contexts.

- **Parent:** <daemon>
- **Default Value:** 60000

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <ttl>60000</ttl>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.15 LBMRD Element "<port>"

Supplies network port to bind the socket for receiving TR traffic from UM contexts. This is the port that a UM context should use when TCP-based TR is configured with the option **resolver_unicast_daemon (context)**. The value contained within the <port>...</port> is an integer between 1 and 65535.

- **Parent:** <daemon>
- **Default Value:** 15380

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <port>15380</port>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.16 LBMRD Element "<interface>"

Specifies the network interface to bind the socket for receiving TR traffic from UM contexts. This is the IP address that a UM context should use when Unicast UDP-based TR is configured with the option **resolver_unicast_daemon (context)**. See **Specifying Interfaces** for methods of specifying the interface within <interface>...</interface>.

If not specified, UM chooses the first interface it finds.

WARNING: It is strongly recommended to specify an interface when running lbmrd, either via the "-i" command-line option, or the <interface> XML tag. Otherwise, UM will select the first interface it finds, potentially 127.0.0.1 (loopback), which is rarely a good choice. Note that CIDR notation can make it easier. For example, "10.0.0.0/8" will match any interface on the 10 network.

- **Parent:** <daemon>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <interface>10.1.1.50</interface>
    ...
  </daemon>
  ...
</lbmrd>
```

16.1.17 LBMRD Element "<activity>"

Interval between client activity checks (in milliseconds)

- **Parent:** <daemon>
- **Default Value:** 60000

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <activity>60000</activity>
    ...
  </daemon>
  ...
</lbmrd>
```

16.2 Dummy lbmrd Configuration File

If no NAT is present, and it is desired to use the XML configuration file for it's '<**daemon**>' contents, a "dummy" NAT configuration should be used.

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    ...
  </daemon>
  <domains>
    <domain name="dummy">
      <network>0.0.0.0/32</network>
    </domain>
  </domains>
  <transformations>
    <transform source="dummy" destination="dummy">
      <rule>
        <match address="0.0.0.0" port="0"/>
        <replace address="0.0.0.0" port="0"/>
      </rule>
    </transform>
  </transformations>
</lbmrd>
```

16.3 Lbmrd DTD file

The DTD file is used to validate the user's configuration file.

```
<!ELEMENT lbmrd (daemon?, domains, transformations)>
<!ATTLIST lbmrd
          version (1.0) #REQUIRED
>
<!ELEMENT daemon (activity|interface|port|ttl|log|resolver_unicast_receiver_socket_buffer|
      resolver_unicast_send_socket_buffer)*>
<!ELEMENT activity (#PCDATA) >
<!ELEMENT interface (#PCDATA) >
<!ELEMENT port (#PCDATA) >
<!ELEMENT ttl (#PCDATA) >
<!ELEMENT log (#PCDATA) >
<!ELEMENT resolver_unicast_receiver_socket_buffer (#PCDATA) >
<!ELEMENT resolver_unicast_send_socket_buffer (#PCDATA) >
<!ELEMENT domains (domain+)>
<!ELEMENT domain (network+)>
<!ATTLIST domain name ID #REQUIRED>
<!ELEMENT network ( #PCDATA )>
<!ELEMENT transformations ( transform+ )>
<!ELEMENT transform ( rule+ )>
<!ATTLIST transform
          source IDREF #REQUIRED
          destination IDREF #REQUIRED
>
<!ELEMENT rule ( match, replace )>
<!ELEMENT match EMPTY>
<!ATTLIST match
          address CDATA #REQUIRED
          port CDATA "*"
>
<!ELEMENT replace EMPTY>
<!ATTLIST replace
          address CDATA #REQUIRED
          port CDATA "*"
>
```

Chapter 17

Packet Loss

This section is about packet loss. It applies primarily to UDP-based network protocols used by UM: LBT-RM, LBT-RU, and UDP-based topic resolution.

(Packet loss also affects TCP-based protocols, but UM has no control over how it is handled by the IP stack. "Packet" loss can also happen with the IPC transport type if source-pacing is selected, but most of this section's discussion doesn't apply to IPC.)

Packet loss is almost always caused when some part of the system is receiving packets at a higher rate than it is able to process them. This might be a router, a switch, a subscribing host's kernel, or the subscribing process itself. This typically results in queuing of incoming packets within the router, switch, kernel, or socket buffer. But queues do not have unlimited size. If the incoming packets exceed the processing speed for too long a period of time, the queue will fill and packets will be dropped.

Packet loss is a fact of life in networks. Some users are able to provision and tune their systems such that they might only lose a few packets per week. Other users routinely live with several lost packets per minute. Many users do not monitor their system for loss and have no idea how frequent it is.

Packet loss is undesirable for many reasons. For reliable protocols (TCP, LBT-RM, etc), detection and retransmission of lost packets introduces significant latency. If packet rates are too high for too long a period of time, the transport protocol can give up trying to recover the lost data. For Streaming, this can cause disconnects or Unrecoverable Loss, where application messages can be lost forever.

Informatica recommends that you:

1. Design your system to prevent packet loss.
2. Configure UM for proper recovery from packet loss.
3. Monitor your network for packet loss.
4. Diagnose the cause of the observed packet loss and fine-tune your design and configuration to better prevent loss and optimize recovery from it.

17.1 Design to Prevent Loss

There are two complementary methods of avoiding packet loss:

- Decrease Packet Flow through Loss Points.
- Increase Efficiency of Packet Consumers.

17.1.1 Decrease Packet Flow through Loss Points

- **Message Batching**. At most loss points, the number of packets is usually more important than the sizes of the packets. 100 packets of 60 bytes each is *much* more burdensome to packet consumers than 10 packets of 600 bytes each. For latency-sensitive applications, consider implementing an Intelligent Batching algorithm.
- **Reduce discards**. Due to the way publishers map topics to Transport Sessions, it is often the case that the receiver will have to discard messages that it hasn't subscribed to. The **LBT-RM transport statistics** structure for each transport type contains the field "lbm_msgs_no_topic_rcved" which counts the number of data messages discarded. Also, the **context statistics** structure contains the field "lbtrm_unknown_msgs_rcved", which also counts data messages discarded.

  For the LBT-RU transport type, you can get similar discards. See the **LBT-RU transport statistics** structure field "lbm_msgs_no_topic_rcved", and the **context statistics** structure field "lbtru_unknown_msgs_rcved".

  For the TCP or LBT-RU transport types, you can often decrease discards by turning on **Source Side Filtering**.

  With Multicast, the Source Side Filtering feature is not possible. So it is usually necessary to change the topic-to-transport session mapping. Ideally, this can be done by taking into account the topic interests of the subscribers. But often it simply means increasing the number of transport sessions. In the case of LBT-RM, increasing the number of multicast groups is preferred, but is often limited by the network hardware. In that case, you can multiply the number of transport sessions by varying the destination port.

17.1.2 Increase Efficiency of Packet Consumers

Here are some methods for increasing the efficiency of subscribers: Use a kernel-bypass driver. For Linux, use Receive Multiple Datagrams. Use Receive Buffer Recycling. For Java and .NET, use Zero Object Delivery.

17.2 UM Recovery of Lost Packets

See Messaging Reliability for a high-level description of message loss as it relates to UM.

UM recovers lost packets at multiple levels:

- **Transport** - TCP, LBT-RU, LBT-RM have low-level handshakes to detect and retransmit lost packets.
- **OTR/Late Join** - independent of transport, OTR and Late Join will recover data, typically after the transport layer is unable to recover.
- **Persistence** - closely-associated with OTR and Late Join, the persistent Store provides a much greater capacity to recover lost data. UM persistence provides a guarantee of message delivery, subject to a set of configurable constraints.

One common goal for most UM use cases is that users want the flow of new messages to continue unimpeded in parallel with recovery efforts of lost packets. Given that packet loss is almost always a result of high packet rates overloading one or more queuing points along a messaging path, the addition of packet recovery efforts can make the overload even worse. "Pouring gasoline on a fire" is an often-repeated metaphor.

Fortunately, packet rate overload tends to be temporary, associated with short-term traffic bursts. That is one reason why the UM lost packet recovery algorithms use time delays. For example: **transport_lbtrm_nak_initial_backoff_interval (receiver)** and **otr_request_initial_delay (receiver)**. By waiting before requesting retransmissions, the burst is allowed some time to subside before we add retransmission to the normal traffic load. These delays do add to latency, but shortening the delay too much risks making the loss worse, which can make the overall latency worse than having a longer delay.

One limiting factor related to data recovery is UM's use of retransmission rate limits. After a short period of severe packet loss due to overload, many receivers will be requesting retransmission. It would make no sense for the initial request delay to successfully bypass the original traffic burst, only to create its own overloading burst of re-transmissions. Older NAK-based systems can get into a positive feedback loop where loss leads to retransmission, which leads to more loss, which leads to more retransmission, etc. Even once new data rates return to normal, networks can be locked into this kind of NAK/Retransmission storm. UM's retransmission rate limiter throttles the retransmission of lost packets over time without worsening the loss.

Another limiting factor related to data recovery is the amount of data which the sender buffers and is available for retransmission. Applications need to continue sending new data while recovery takes place. Since the buffer is of limited size, older buffered messages will eventually be overwritten with new messages.

For streaming applications, these buffers are held in memory, and the sizes are usually measured in megabytes. For persistent applications, the Store writes its buffer to disk, allowing for buffer sizes orders of magnitude larger than memory-based buffers. But even the Store's disk-based buffer is of finite size, and is susceptible to being overwritten if it takes too long to recover data.

Note that the **Receiver Paced Persistence (RPP)** feature seeks to maximize the reliability of messaging by allowing the publisher to be blocked from sending rather than overwriting unacknowledged data.

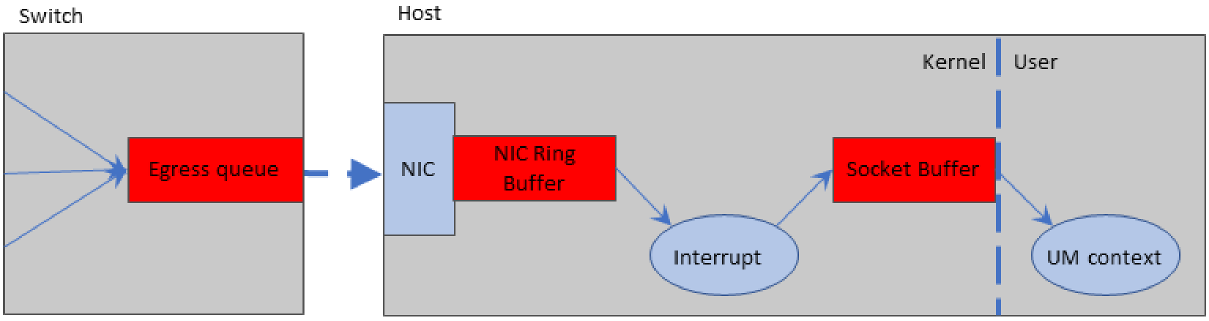
Given these limiting factors for data recovery, a sufficiently-overloaded network can reach a point where lost data can no longer be recovered, a situation called "unrecoverable loss".

Finally, it is very important for UM users to make use of UM's extensive monitoring capabilities. Since UM does a good job of recovering lost packets, you may be experiencing high latency spikes without knowing it. And recovered loss today can be a warning of unrecoverable loss tomorrow. User are strongly advised to monitor transport statistics and pay special attention to receivers that repeatedly experience loss. Even if that loss is successfully recovered, you should diagnose and treat the loss before it gets worse and becomes unrecoverable.

See **Monitoring** for more information.

17.3 Packet Loss Points

There are just a few common points at which packets are normally lost:



The red buffers/queues are the most common locations where packets are typically lost during a packet burst.

17.3.1 Loss: Switch Egress Port

The switch egress port can come under pressure if data flows from multiple sources need to be merged onto a single outgoing link. The outgoing link can be an internal trunk or communication link connecting two pieces of network equipment, but more typically it is link to a destination host.

It is easy to understand how loss can happen here. Suppose three remote hosts are sending UDP data streams at the destination host. If each stream is carrying 0.5 gigabit/sec of throughput, the switch needs to send 1.5 gigabit/sec over a 1 gigabit link, a clear overload. If this is a very short-term burst, the egress queue will hold the data until the incoming data flows subside and the outgoing port can get caught up. But if the burst lasts too long and the egress queue fills, the switch has no choice but to drop packets.

Note that the switch will not count these drops as "errors". There is a separate drop counter which should be examined to diagnose switch egress port loss.

MITIGATION

The only solution is to reduce the packet rate being sent to the destination host. See Decrease Packet Flow through Loss Points.

17.3.2 Loss: NIC Ring Buffer

As packets are received by the host's NIC (Network Interface Card), they are copied into host memory in a structure called the Receive Ring Buffer. The NIC interrupts the OS, which has the responsibility to unload the packet buffers from the Ring Buffer. If the incoming packet rate is faster than the OS can unload the Ring Buffer, it will fill and packets will be dropped.

Normally, the kernel is able to service NIC interrupts without any trouble. However, there is one situation which can put the Ring Buffer under pressure: When multiple processes on the host are subscribed to the same multicast stream, the kernel must replicate and deliver the packets to each process. For a small number of processes (5-10), the kernel will still be able to keep up with the incoming packets.

However, as companies consolidation servers by moving to large, many-core hosts (often virtualized), we see the same multicast stream subscribed to by increasing numbers of processes on the same physical server. We have seen NIC Ring Buffer loss (also called "overrun") with as few as 15 processes subscribed to a heavy stream of multicast packets.

(Note that this is generally only a problem for multicast. With Unicast data distribution to many recipients, the source essentially does the packet replication work. The receive-side work for the kernel for each unicast packet is minimal.)

MITIGATION

Users should maximize the size of the NIC's Receive Ring Buffer. For many NICs, the size of the ring buffer is configured by the number of receive descriptors. This should be set to the maximum allowable value.

The mitigators listed in Loss: Switch Egress Port will also help this problem by reducing the incoming packet rate.

Another solution is to spread processes across more physical hosts. This has the additional advantage of reducing latency, since multicast replication within a host must be done in software by the kernel and is serial in nature, whereas replication in the network is done by specialized hardware in parallel.

Another possible solution involves the use of the DRO as the primary receiver of the multicast data, which then re-publishes it on the host using the IPC transport. This has the disadvantage of introducing some additional latency since the messages must be received by the DRO and then forwarded to the application receivers. It also requires separating the applications to their own Topic Resolution Domains (TRDs).

Onload users can get more information at **Solarflare Tips**.

17.3.3 Loss: Socket Buffer

The Socket Buffer represents the interface between the OS kernel and the user process. Received data is transferred from the NIC Ring Buffer to the destination Socket Buffer(s). The Socket Buffers are then emptied by the application process (in this case, the UM Context Thread). Socket buffer sizes are configurable, according to the transport type. For example, see **transport_lbtrm_receiver_socket_buffer (context)**.

The TCP protocol is designed to ensure that the socket buffer cannot be overflowed. However, UDP-based protocols (LBT-RU and LBT-RM) are susceptible to socket buffer overflow, which leads to datagram loss.

MITIGATION

All of the mitigators listed Loss: NIC Ring Buffer will help this problem by reducing the incoming packet rate.

An obvious mitigator is to increase the sizes of the receive socket buffers. Informatica usually recommends at least 8MB for UDP-based protocols. But this only works if the problem is related to short-term traffic bursts. Simply increasing the size of the buffer will not avoid loss if the *average* message rate exceeds the average consumption and processing rate of the receiving program.

A very useful method for mitigating socket buffer loss is to increase the efficiency of the receiving application. The Receive Multiple Datagrams can increase that efficiency without sacrificing latency.

Also, the Transport Services Provider (XSP) feature can help by splitting the work of unloading multiple sockets across multiple threads.

17.3.4 Loss: Other

The three loss locations described above are all related to high packet rates causing fixed-sized packet buffers to overflow. These represent by far the most common reasons for packet loss. However, it is possible that you will experience loss that cannot be diagnosed to those three causes.

For example, we have seen reports of NIC hardware malfunctioning such that most packets are successfully received and delivered, but some percentage of packets fail. At least one user reported that a misconfigured router "flapped" a route, resulting in periodic, short-term loss of connectivity between two sub-networks. We have seen a case where the use of kernel bypass drivers for high-performance NICs (specifically Solarflare) can cause multicast deafness if both accelerated and non-accelerated processes are run on the same host. We have even seen a case where replacing the Ethernet cable between a host and the switch resolved packet loss.

It is not possible to have a step-by-step diagnostic procedure which will pinpoint every possible cause of packet loss. The techniques described in this document should successfully diagnose a large majority of packet loss causes, but nothing can replace your infrastructure network engineers expertise at tracking down problems.

17.4 Verifying Loss Detection Tools

The preceding techniques for mitigating loss are best deployed after you have identified the type of loss. Unfortunately, we have found that the tools available to detect and identify the location of loss to be problematic. Informatica does not provide such tools, and does not follow the market for such tools to find a reliable supplier.

However, we have a starting point that has given us some measure of success in diagnosing the loss points. It is important that you try out these tools to verify that they properly detect the different types of loss. In order to verify them, you need to be able to reproduce on demand loss at each of the points: switch, NIC, and socket buffer.

Fortunately, this is reasonably easy using the `msend` and `mdump` tools provided in the "mtools" package offered by Informatica free of charge. Download the mtools package from `https://community.informatica.com/solutions/informatica_mtools` The source files for `msend` and `mdump` are provided, as well as pre-built binaries for most major platforms.

Informatica recommends verifying your loss diagnosis tools *before* you have a serious loss event that disrupts your application system, preferably before your system goes into full production usage. Periodically running and recording the results of these tools during normal operation will make it possible to diagnose loss after the fact. Detecting and identifying non-severe (recoverable) loss can be used to prevent serious (unrecoverable) loss events in the future.

17.4.1 Prepare to Verify

1. Download and install mtools on two hosts, designated "sender" and "receiver". Informatica recommends that the hosts be "bare metal" (not virtual machines), and that they be connected to the same switch. This minimizes the chances that the verification tests will cause any disruption to normal operation.

2. Contact your system and network administrators and set up some time that they can work with your during the verification process. They will need to perform operations that you probably do not have the ability to do.

3. Have the network administrator allocate a multicast group that you can use for this test. That multicast group should be otherwise unused in your organization. Warn the administrator that you will be pushing intense traffic bursts between the two hosts.

17.4.2 Verifying Switch Loss

A possible Unix command that a network administrator could use is:

`snmpwalk -v 1 -c public` *SWITCH_ADDR* `IF-MIB::ifOutDiscards`

Note that the above community string ("public") is probably not enabled; the network administrator will know the appropriate value. Ideally, the network administrator would run that command every 5 or 10 minutes, logging to a file, with a time stamp. If this log file could be shared read-only to the project groups, they can time-correlate any unusual application event with loss reported by the switch.

To verify that you properly detect switch loss, follow these steps:

1. Work with your system and network administrators to **enable** Ethernet flow control in both the switch port and the NIC.

2. Use the above `snmpwalk` command (or equivalent) to record the current drop counts for the switch ports.

3. On the receiving host, run 30 copies of the following command:
   `mdump -q` *MCAST_ADDR* `12000` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the receiving host.

4. On the sending host, run the following command:
   `msend -5` *MCAST_ADDR* `12000 15` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the sending host.

5. When the test completes, use the `snmpwalk` command again (or equivalent) to record another set of drop counters. The receiving host's drop count should be larger.

This test works by making the receiving host's kernel work very hard for each received datagram. It should be unable to keep up. (If you don't see any drops caused by the test, try doubling the number of copies of `mdump` on the receiving host.) The Ethernet flow control settings on the NIC and switch will prevent NIC loss in its ring buffer by slowing down the switch's egress port. Thus, the switch's egress queue will fill and should overflow.

17.4.3 Verifying NIC Loss

Unix

On some Unix systems, the "ifconfig" command will accurately report receive overrun on the NIC. For example:

```
ifconfig eth0
```

But in many Unix systems, the values reported by "ifconfig" remain at zero, even when the NIC has in fact overrun its receive ring buffer. We recommend also trying the "ethtool" command. For example:

```
ethtool -s eth0
```

Windows

To the best of our knowledge, there is no standard Windows tool for detecting NIC loss. Some drivers might provide that information from the interface control panel. Otherwise, you might need to download a management application from the NIC or system vendor.

If you know of a widely-available method to detect NIC overrun on Windows, please let us know at our **D↩ LMessagingBuilds** email account on informatica.com (that awkward wording used to avoid spam address harvesters).

To verify that you properly detect NIC loss, follow these steps:

1. Work with your system and network administrators to **disable** Ethernet flow control in both the switch port and the NIC.

2. Use your NIC loss tool to get the current receive overrun count.

3. On the receiving host, run 30 copies of the following command:
   `mdump -q` *MCAST_ADDR* `12000` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the receiving host.

4. On the sending host, run the following command:
   `msend -5` *MCAST_ADDR* `12000 15` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the sending host.

5. When the test completes, use the NIC loss tool again to record the receive overrun count.

This test works by making the receiving host's kernel work very hard for each received datagram. It should be unable to keep up. (If you don't see any drops caused by the test, try doubling the number of copies of `mdump` on the receiving host.) The lack of Ethernet flow control means that the switch will send the packets at full line rate, which should overflow the NIC ring buffer.

17.4.4 Verifying Socket Buffer Loss

On most systems, the `netstat` command can be used to detect socket buffer overflow. For example:

```
netstat -s
```

Look in the UDP section for "receive errors". This normally represents the number of datagrams dropped due to the receive socket buffer being full.

Note that Windows prior to version 7 does not increment that field for socket buffer overflows. If you have pre-↩ Windows 7, we don't know of any command to detect socket buffer overflow.

To verify that you properly detect socket buffer overflow, follow these steps:

1. Use `netstat -s` to get the current receive error count.

2. On the receiving host, run a single copy of the command:
   `mdump -q -p1000/5` *MCAST_ADDR* `12000` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the receiving host.

3. On the sending host, run the following command:
   `msend -5 -s2200` *MCAST_ADDR* `12000 15` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the sending host.

4. When the test completes, use `netstat -s` again to get the new receive error count.

This test works by introducing a short sleep in the "mdump" command between reception of datagrams. This causes the socket buffer to overflow.

17.5 TCP Disconnections

The TCP protocol is generally very good at recovering from periods of packet loss. However, if no packets at all can get through for an extended period, TCP will reach its retry limit and disconnect. This can affect the TCP transport and UIMs.

This can happen if there is a network fault that lasts for an extended period.

Another potential root cause of TCP disconnections can be a `stateful firewall`. These devices monitor traffic on active TCP connections and will time out idle connections if no packets are sent for a configurable period, typically measured in minutes. The firewall typically does not inform the hosts using the TCP connection, so the hosts think the connection is still valid. However, when the sending host tries to transmit, the firewall will simply drop the packets, leading to the sender timing out the TCP connection and disconnecting.

Note that the other side of the connection often does not detect this disconnection, leading to a `half-open` connection. For the TCP transport, this can prevent a receiver from restarting topic resolution, which can be necessary for the receiver to recover from a temporary outage.

One solution to this problem is to enable TCP keepalives by setting **transport_tcp_activity_method (receiver)** to "SO_KEEPALIVE" and **transport_tcp_activity_timeout (source)** to a greater-than-zero value for TCP transports, and setting **request_tcp_activity_timeout (context)** and **response_tcp_activity_timeout (context)** to greater-than-zero values for UIMs. These keepalives prevent the firewall from timing out a quiet TCP connection, or if the connection is lost for some other reason, will inform both hosts in a timely fashion.

Chapter 18

UM Glossary

18.1 Glossary A

ABI - Application Binary Interface

The execution-time interfaces presented by one software system, generally in the form of a dynamic (shared) library, for use by other software systems. ABIs are generally considered to be in the realm of binary, compiled code, not source code. Two versions are considered ABI compatible if the dynamic libraries can be used interchangeably by an application without the need to rebuild or relink that application. See also API.

ACK - Acknowledge

Generally, a control message which acknowledges some event or condition. Within the context of Ultra Messaging, it is often used to refer to a persistence control message sent by a subscriber to the Persistent Store to indicate that it has completed processing of a given data message. See Persistence.

ACE - Access Control Entry

A filter specifier to control which topics are allowed to transit a DRO portal. One or more ACEs make up an Access Control List (ACL). See **Access Control Lists (ACL)**.

ACL - Access Control List

A method used by the DRO to control which topics are allowed to transit a DRO portal. An ACL consists of one or more Access Control Entries (ACE). See **Access Control Lists (ACL)**.

ActiveMQ

The name of an open-source JMS-oriented messaging system. The Ultra Messaging UMQ product contains an enhanced form of ActiveMQ to provide queuing semantics and a JMS API. See **UMQ Overview**.

AMQP - Advanced Message Queuing Protocol

An open standard messaging wire protocol. See `Wikipedia's write-up` for more information on AMQP. The UMQ product makes use of AMQP to provide interoperability between Ultra Messaging and ActiveMQ. See **UMQ Overview**.

API - Application Programming Interface

The callable functions, classes, methods, data formats, and structures presented by one software system for use by other software systems. APIs are generally considered to be in the realm of source code, not compiled binaries. APIs are generally documented, and can be extended from one version to the next. Two versions are considered API compatible if the application can be built against either version interchangeably without the need to modify the source code. Ultra Messaging has APIs available for the C, Java, and .NET (C#) programming languages. For example, **lbm_context_create()** is part of the C API. See also ABI.

18.2 Glossary B

BOS - Beginning Of Stream

An event delivered to a receiver callback indicating that the link between the source and the receiver is now active. Be aware that in a deployment that includes the DRO, it may only indicate an active link between the receiver and the local router portal, not necessarily full end-to-end connectivity. See also EOS

Broker

A daemon which mediates the exchange of messages. In the context of Ultra Messaging, it refers to the ActiveMQ daemon which implements the queuing functionality and JMS. See Queuing.

Busy Waiting

Also known as "busy looping" and "polling". A method of a thread to wait for a real-time event by testing for the event in a tight loop without giving up the CPU. Typically leads to the thread consuming 100% of a CPU core.

18.3 Glossary C

CIDR - Classless Inter-Domain Routing

Generally, CIDR refers to the division of a 32-bit IPv4 address between network and host parts. In the context of Ultra Messaging, CIDR notation can be used to ease the specification of host network interfaces. See **Specifying Interfaces**.

Context

Within the context of Ultra Messaging, a context is an object which functions conceptually as an environment in which UM runs. Context is often abbreviated as "ctx". See Context Object.

Crybaby Receiver

Within the context of a NAK-based protocol (like LBT-RM or LBT-RU), a crybaby receiver is one that experiences sustained loss even during periods of normal traffic load. This might be due to equipment malfunction (e.g. a lossy NIC) or software malfunction (application which can't keep up with normal traffic). See NAK Suppression.

CTX - Context

Within the context of Ultra Messaging, a context is an object which functions conceptually as an environment in which UM runs. See Context Object.

18.4 Glossary D

DBL - Datagram Bypass Layer

A kernel-bypass driver that accelerates sending and receiving UDP traffic and operates with Myricom 10-Gigabit Ethernet adapter cards for Linux and Windows. See **Myricom Datagram Bypass Layer (DBL)**.

Deafness

An informal term for a problem where a subscriber for a topic is not receiving messages sent by a publisher for that topic.

Delivery Confirmation

An optional event generated by a persistent subscriber's receiver and delivered to a persistent publisher's source to indicate that the subscriber has completed processing of a message. See Persistence.

Delivery Controller

An instance of the receive-side "topic layer" within the UM software stack. See UM Software Stack.

DLQ - Dead Letter Queue

With queuing, the Dead Letter Queue (DLQ) is a destination for messages that cannot be delivered to a receiver. See **Dead Letter Queue**.

DRO - Dynamic Routing Option

The name of an Ultra Messaging option which provides routing of messages different Topic Resolution Domains (TRDs). See DRO.

Dynamic Routing Option (DRO)

The name of an Ultra Messaging option which provides routing of messages different Topic Resolution Domains (TRDs). See DRO.

18.5 Glossary E

EOS - End Of Stream

An event delivered to a receiver callback indicating that the link between the source and the receiver is deleted. Be aware that in a deployment that includes the DRO, it may only indicate a deleted link between the receiver and the local router portal, not necessarily a full end-to-end link. See also BOS

Event Queue

Within the context of Ultra Messaging, an event queue object is a serialization queue structure and execution thread for delivery of other objects' events. Event queue is often abbreviated as "evq". See Event Queue Object.

EVQ - Event Queue

Within the context of Ultra Messaging, an event queue object is a serialization queue structure and execution thread for delivery of other objects' events. See Event Queue Object.

18.6 Glossary F

Flight Size

The number of messages that a persistent publisher can have outstanding that are not stable. A persistent publisher generally limits the number of unstable messages it can have outstanding, and may block further attempts to send until some outstanding messages become stable. See Persistence. See also Stability.

Fragmentation Size

The splitting of a large application message into multiple pieces. There are two forms of fragmentation: UM fragmentation (done by UM) and IP fragmentation (done by the operating system). See Message Fragmentation and Reassembly.

18.7 Glossary G

Gateway

An early version of a message router, replaced as of version 6.10 with the DRO. See Dynamic Routing Option (DRO).

18.8 Glossary H

HF - Hot Failover

A form of redundancy in which multiple instances of a publisher send the same messages at the same time to subscribers, which select for application delivery the first copy received. If one publisher instance fails, the subscribers are able to continue operation receiving from the remaining publisher. See Hot Failover (HF).

HFX - Hot Failover eXtended

An extended form of Hot Failover. Note: HFX is deprecated. See Hot Failover Across Multiple Contexts (HFX).

HRT - High Resolution Timestamp

A feature that leverages the hardware timestamping function of certain network interface cards to measure sub-microsecond times that packets are transmitted and received. See High-resolution Timestamps.

18.9 Glossary I

IP Fragmentation

The Operating System function of splitting UDP datagrams into `MTU-sized` packets. See Message Fragmentation and Reassembly.

IPC - InterProcess Communication

Generally, the term simply refers to any of several mechanisms by which an operating system allows processes to communicate or share data. Within the context of Ultra Messaging, LBT-IPC specifically refers to the shared memory transport type. A source configured for LBT-IPC can only pass messages to receivers running on the same machine (or virtual machine). See Transport LBT-IPC.

18.10 Glossary J

Jitter

The amount of variation from the average, usually latency. High jitter can either mean frequent small variations, or infrequent large variations. Large latency outliers are undesirable, even if rare.

JMS - Java Message Service

A standardized API for Java applications to send and receive messages. Ultra Messaging's UMQ product allows limited interoperability between applications using UM and applications using JMS. See **JMS**.

JNI - Java Native Interface

A method by which Java code can invoke code written in C.

18.11 Glossary K

Kernel-Bypass Driver

A device driver software package, normally supplied by a hardware vendor, which provides a user-space library and API for accessing the hardware without transitioning into the kernel. Some examples: Solarflare's Onload driver, Myricom's DBL driver, Cisco's "ExaSOCK" driver, Voltaire's VMA driver. See Datagram Max Size and Network MTU for related information.

18.12 Glossary L

LBM - Latency Busters Messaging

An old name of the Ultra Messaging product line. Superseded by UM. "LBM" is sometimes used to refer to the streaming product. That use is superseded by "UMS". The abbreviation "lbm" lives on in various parts of the UM API, and was kept for backwards compatibility.

LBT - Latency Busters Transport

Usually used as a prefix for a specific transport type: LBT-RM, LBT-RU, LBT-IPC, and LBT-SMX. See **transport (source)**.

LJ - Late Join

A function by which a subscriber can create a receiver for a topic, and is able to retrieve one or more messages sent to that topic prior to the receiver being created. See Late Join.

LJIR - Late Join Information Request

A type of control message sent by a receiver to a source to request an Late Join Information control message. See Late Join.

18.13 Glossary M

MIM - Multicast Immediate Message

Alternate send method which makes use of a pre-configured LBT-RM transport which is shared by all like-configured applications. The "immediate" means that messages may be sent to arbitrary topic names without the creation of source objects. See Multicast Immediate Messaging. See also glossaryuim.

18.14 Glossary N

NAK - Negative AcKnowledgement

A type of control message sent by a receiver using LBT-RM or LBT-RU transports. Sent when packet loss causes a sequence number gap in received messages, the NAKs specify which sequence numbers are missing and request retransmission. See **Transport LBT-RM Reliability Options**.

NCF - NAK ConFirmation

A type of control message sent by a source using LBT-RM transport. The LBT-RM protocol requires a source send an NCF if it receives a NAK for which it is not willing to send a re-transmission. See **LBT-RM Source Ignoring NAKs for Efficiency**.

NAK Storm

A network condition in which packet loss triggers packet recovery operations for a NAK-based protocol (like L↩BT-RM and LBT-RU), and the packet recovery operations themselves induce further packet loss. This can lead to a self-reinforcing feedback loop in which the packet recovery operations trap the protocol in an unbounded loss condition. See What Is a NAK Storm? for details and prevention.

NIC - Network Interface Card

A part of a computer which connects to one or more network cables and provides packet-level communication to the operating system.

18.15 Glossary O

Onload

A kernel-bypass driver that accelerates sending and receiving network traffic and operates with Solarflare Ethernet adapter cards for Linux. See **Solarflare Onload**.

OpenOnload

The open-source version of Onload, a kernel-bypass driver that accelerates sending and receiving network traffic and operates with Solarflare Ethernet adapter cards for Linux. See **Solarflare Onload**.

OpenSSL - Open Secure Sockets Layer

A library which provides encryption services. OpenSSL is used by Ultra Messaging's encryption feature. See https://www.openssl.org for general information about OpenSSL. See Encrypted TCP for information about Ultra Messaging's encryption feature.

OTID - Originating Transport IDentifier

Control information which uniquely identifies a source object within a UM network. See **More About Proxy Sources and Receivers**.

OTR - Off-Transport Recovery

A method by which are lost and are not recoverable by the source transport can be recovered by UIMs using a method similar to Late Join. See Off-Transport Recovery (OTR).

18.16 Glossary P

Pacing

In messaging, pacing refers to the method by which the send rate is controlled (or not controlled). In general, there is "source pacing", where the source determines the rate of message transmission, or "receiver pacing", where the slowest receiver of a source can limit the rate of message transmission. See Transport Pacing.

PCRE - Perl Compatible Regular Expressions

An open-source library which closely implements the Perl 5 regular expression language. UM uses PCRE for wildcard receiver pattern matching. See Wikipedia's write-up for information on PCRE. See also UM Wildcard Receivers.

PDM - Pre-Defined Messages

A message encoding scheme based on integer field identifiers for structured messages can be assembled and sent by applications. Includes field types and performs data marshaling across different CPU architectures. See Pre-Defined Messages. See also SDM.

Persistence

One of the basic Messaging Paradigms supported by UM (the other two are Streaming and Queuing). Persistence, sometimes called "durable" or "guaranteed" messages, saves messages sent by a publisher in non-volatile storage so that subscribers can recover missed messages under a variety of failure scenarios. If multiple subscribers exist for the same topic, each subscriber will get all messages sent by the publisher. See Persistence.

PGM - Pragmatic General Multicast

A standards-based protocol for reliable multicast. Ultra Messaging's "LBT-RM" protocol is inspired by PGM. See Transport LBT-RM for a list of differences between LBT-RM and PGM..

Pinning

To achieve the highest performance, users should make use of "core pinning", also called "task setting" or "setting thread affinity". This is where time-critical threads are assigned to execute on a specific set of one or more CPU cores. Without pinning, the operating system will sometimes migrate a thread from one NUMA zone to another. This introduces significant latency and jitter. Core pinning prevents this from happening. See Core Pinning.

Portal

An interface to the DRO. A DRO portal can either be an endpoint portal (interfaces with a Topic Resolution Domain), or a peer portal (interfaces with another DRO). See **DRO Portals**.

PTP - Precision Time Protocol

A protocol used to synchronize clocks throughout a computer network. Used by some NICs to synchronize host clocks (e.g. Solarflare). See `Wikipedia's write-up` for more information.

Pub/Sub - Publish / Subscribe

A model of messaging passing in which the publisher (sender) does not keep track of the subscribers (intended recipients) of messages. Instead, the messages carry metadata (topic name) in which the subscribers express interest, and the underlying messaging software forwards messages to the subscribers based on that interest.

18.17 Glossary Q

Queuing

One of the basic Messaging Paradigms supported by UM (the other two are Streaming and Persistence). Queuing supports "load balancing" whereby published messages can be distributed across a set of subscribers such each message is only handled by one of the subscribers. See Queuing.

18.18 Glossary R

RCV - Receiver

Within the context of Ultra Messaging, a receiver is an object used to subscribe to a topic. "Receiver" is sometimes used to refer generally to an entire subscribing application. See Receiver Object. See also Wildcard Receiver.

Receiver

Within the context of Ultra Messaging, a receiver is an object used to subscribe to a topic. "Receiver" is sometimes used to refer generally to an entire subscribing application. Receiver is often abbreviated as "rcv". See Receiver Object. See also Wildcard Receiver.

Registration

When a publisher creates a persistent source, that source must register with the configured persistent Stores before it can start sending messages. This registration prepares the persistent Store and the source to co-operate in the transfer of persisted messages. Likewise, when a subscriber creates a persistent receiver, that receiver must register with the configured persistent Stores before it can start receiving messages. See Persistence.

Request Port

Also known as "UIM Port". TCP port that a context listens to for incoming UIM traffic. See Unicast Immediate Messaging for general information on UIM.

RM - Reliable Multicast

A shortening of "LBT-RM". The Ultra Messaging protocol and implementation in which user messages sent via Multicast UDP are monitored for loss, and retransmissions are arranged to recover loss. See Transport LBT-RM.

RPP - Receiver-Paced Persistence.

A form of persistence in which a publisher can be blocked from sending if receivers are having trouble keeping up with the message rate. See Persistence. See also SPP.

Router

Within the context of Ultra Messaging, "\ref umrouter" generally refers to the daemon within the DRO. See Dynamic Routing Option (DRO).

RSA - Rivest, Shamir, and Aleman

A public-key cryptosystem developed by Ron Rivest, Adi Shamir, and Leonard Adleman. Included in the Open↩ SSL library used by Ultra Messaging's encryption feature. See Encrypted TCP.

RU - Reliable Unicast

A shortening of "LBT-RU". The Ultra Messaging protocol and implementation in which user messages sent via Unicast (point-to-point) UDP are monitored for loss, and retransmissions are arranged to recover loss. See Transport LBT-RU.

RX - Re-transmission

Depending on the context, RX can either mean the messages retransmitted by the LBT-RM and LBT-RU transport protocols (e.g. in transport statistics), or it can mean the messages recovered via the Persistent Store or Late Join.

18.19 Glossary S

SDM - Self-Describing Messages

A message encoding scheme based on keyword-value pairs for structured messages can be assembled and sent by applications. Includes field types and performs data marshaling across different CPU architectures. See Self Describing Messaging. See also PDM.

SIR - Source Information Record

A type of control message used by the SRS to advertise sources to contexts. See TCP-Based Topic Resolution Details.

SM - Session Message

A type of control message used by the LBT-RM protocol to keep a Transport Session alive. See **Transport LBT-RM Operation Options**.

SNMP - Simple Network Management Protocol

A standardized protocol by which computers and network equipment can be monitored and managed from a central point (management station). SNMP is also the name of an Ultra Messaging option which makes UM application usage statistics available for monitoring by a standard SNMP management station.

Source

Within the context of Ultra Messaging, a source is an object used to send messages to a topic. "Source" is sometimes used to refer generally to an entire publishing application. Source is often abbreviated as "src". See Source Object.

SPP - Source-Paced Persistence.

A form of persistence in which a publisher is allowed to continue sending at its natural rate, even if one or more receivers are falling behind to the point that the message repository's oldest messages are overwritten, leading to unrecoverable loss. See Persistence.

SRC - Source

Within the context of Ultra Messaging, a source is an object used to send messages to a topic. "Source" is sometimes used to refer generally to an entire publishing application. See Source Object.

SRS - Stateful Resolver Service

A daemon which provides TCP-based Topic Resolution. See Topic Resolution Description.

SRI - Source Registration Information

A type of control message used to communicate persistence information between persistent publishers and subscribers. A subscriber of persistent messages needs an SRI to successfully register with a persistent Store. See Persistence.

Stability

The state that a persistent publisher's sent message has been successfully persisted in the persistent Store. In the time between message transmission and message stability, the message is at risk of being lost. The term is also used to refer to the source event delivered to a publishing application to indicate a message's stability. See Persistence. See also Flight Size.

Store

A shortening of "persistent Store". An Ultra Messaging component which works with persistent sources and receivers to record messages, and also deliver previously-recorded messages for recovery. The UMP and UMQ products include the persistent Store; the UMS product does not. See Persistence.

Streaming

One of the basic Messaging Paradigms supported by UM (the other two are Persistence and Queuing). Streaming requires that the publisher and subscriber be running at the same time for messages to be delivered. Messages are not saved to non-volatile storage. If multiple subscribers exist for the same topic, each subscriber will get all messages sent by the publisher. See Streaming.

Strong RegID

When Persistence is configured with a non-zero activity timeout, the Store prevents simultaneous use of a registration ID by two processes. The registration ID is said to be "strong". See **Activity Timeout and State Lifetimes**.

18.20 Glossary T

TIR - Topic Information Record

A type of topic resolution control message used by a source to advertise its details. Subscribers use TIRs to discover and connect to sources of interest. See Topic Resolution Overview.

Topicless

Related to Immediate Messaging, a "topicless" message is one that has no topic associated with it.

TNWG - Twenty Nine West Gateway

A historic name for the DRO. The name was changed to DRO in UM version 6.0, but the older name of the executable file (`tnwgd`) was retained for backwards compatibility.

TQR - Topic Query Record

A type of topic resolution control message used by a receiver to discover sources of interest. Publishers use TQRs to trigger the sending of TIRs. See Topic Resolution Overview.

TR - Topic Resolution

The protocol used by Ultra Messaging components to exchange information about available topics and topic interest. See Topic Resolution Overview. See also TIR and TQR.

Transport Session

A specific run-time instance of a transport type to carry application messages. The Transport Session can be thought of as a communications channel. As a publishing application creates sources, it maps those sources onto Transport Sessions. A Transport Session is fairly resource-intensive, so it is frequently the case that many sources are mapped to each Transport Session.

TRD - Topic Resolution Domain

A group of Ultra Messaging applications and UM components which communicate with each other directly, not through a DRO. Specifically, it refers to those applications and components which directly exchange Topic Resolution control messages. Applications in different TRDs are not able to communicate with each other unless one or more DROs are used to interconnect the TRDs. See Topic Resolution Domain.

TSNI - Topic Sequence Number Information

A type of control message sent by a source to assist in the detection and recovery from certain types of packet loss. See Loss Detection Using TSNIs.

18.21 Glossary U

UIM - Unicast Immediate Message

Alternate send method which makes use of pre-configured TCP transports. The "immediate" means that messages may be sent to arbitrary topic names without the creation of source objects. Sending a UIM bypasses Topic Resolution, so the calling application must specify the address information for the intended recipient. Because of this, the UIM feature is rarely used directly by user applications. However, Ultra Messaging uses UIMs internally for many of its control messages. See Multicast Immediate Messaging. See also glossarymim.

UIM Port - Unicast Immediate Messaging Port

Also known as "Request Port". TCP port that a context listens to for incoming UIM traffic. See Unicast Immediate Messaging for general information on UIM.

ULB - Ultra Load Balance

A feature of the Ultra Messaging UMQ product which provides a limited subset of queuing semantics without the use of a central message broker. ULB is generally used to provide high-speed load balancing of UM messages. In the Pub/Sub model, if multiple subscribers create receivers for the same topic, each subscriber will receive a copy of every message sent. In the Queuing model, the messages are *distributed* to the multiple subscribers, with each message only being acted on by one of those subscribers. See **Ultra Load Balancing (ULB)**.

UM - Ultra Messaging

The name of the Informatica messaging middleware product line. UM is based on the pub/sub model of message passing, which allows the components of distributed applications to communicate. Note that Ultra Messaging is registered trademark of Informatica, LLC.

UM Fragmentation

The UM function of splitting large application messages into datagrams. See Message Fragmentation and Reassembly.

UM Router

Within the context of Ultra Messaging, "\ref umrouter" generally refers to the daemon within the DRO. See Dynamic Routing Option (DRO).

UMCache - Ultra Messaging Cache

The name of an Ultra Messaging option which provides a limited degree of message storage and retrieval.

UMDS - Ultra Messaging Desktop Services

The name of an Ultra Messaging option which consists of a server daemon and a set of client libraries which provides simplified access to an Ultra Messaging network.

UME - Ultra Messaging, Enterprise edition

An old name of the UMP product. Superseded by UMP. The abbreviation "ume" lives on in various parts of the UM API, and was kept for backwards compatibility.

UMM - Ultra Messaging Manager

A component of UM which allows users to centrally edit, store, and distribute configuration information to distributed applications. See the `UM Manager Guide`.

UMP - Ultra Messaging, Persistence edition

An Ultra Messaging product which supports message Streaming and Persistence. The term is sometimes used to refer specifically to the persistence function. See Persistence.

UMQ - Ultra Messaging, Queuing edition

An Ultra Messaging product which supports message Streaming, Persistence, and Queuing. The term is sometimes used to refer specifically to the queuing function. See Queuing.

UMS - Ultra Messaging, Streaming edition

An Ultra Messaging product which supports message Streaming. The term is sometimes used to refer specifically to the streaming function.

18.22 Glossary V

VMA - Voltaire Messaging Accelerator

A kernel-bypass driver that accelerates sending and receiving UDP traffic and operates with Mellanox 10-↩ Gigabit Ethernet and Infiniband adapter cards for Linux. (The software used to be owned by a company called Voltaire, which was acquired by Mellanox.) See **UD Acceleration for Mellanox Hardware Interfaces**.

18.23 Glossary W

Weak RegID

When Persistence is configured with a zero activity timeout, the Store does not prevent simultaneous use of a registration ID by two processes. The registration ID is said to be "weak". See **Activity Timeout and State Lifetimes**.

Wildcard Receiver

An object created by an application using the UM API to subscribe to a group of topics based on a Regular Expression pattern match. See UM Wildcard Receivers. See also PCRE. See also Receiver.

18.24 Glossary X

XSP - Transport Services Provider

An object created by a subscribing application to control the threading of message reception. See Transport Services Provider Object.

18.25 Glossary Z

ZOD - Zero Object Delivery

Feature which allows a Java or .NET subscribers to have received messages delivered without per-message object creation. This is more efficient than creating objects with each received message, and also avoids garbage collection. See Zero Object Delivery.