Informatica Ultra Messaging Persistence Edition (Version 6.7.1)

Concepts Guide

Informatica Ultra Messaging Persistence Edition Concepts Guide

Version 6.7.1
August 2014

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

Part Number: UMP-CPT-67100-0001

Table of Contents

Preface

This document introduces important fundamental design concepts behind Ultra Messaging high performance message streaming. Understanding these concepts is important to software developers designing and writing application code that uses the Ultra Messaging Application Programming Interface (API). For information about parallel persistence see *The Ultra Messaging Guide for Persistence.*

Informatica Resources

Informatica My Support Portal

As an Informatica customer, you can access the Informatica My Support Portal at http://mysupport.informatica.com.

The site contains product information, user group information, newsletters, access to the Informatica customer support case management system (ATLAS), the Informatica How-To Library, the Informatica Knowledge Base, Informatica Product Documentation, and access to the Informatica user community.

Informatica Documentation

The Informatica Documentation team takes every effort to create accurate, usable documentation. If you have questions, comments, or ideas about this documentation, contact the Informatica Documentation team through email at infa_documentation@informatica.com. We will use your feedback to improve our documentation. Let us know if we can contact you regarding your comments.

The Documentation team updates documentation as needed. To get the latest documentation for your product, navigate to Product Documentation from http://mysupport.informatica.com.

Informatica Web Site

You can access the Informatica corporate web site at http://www.informatica.com. The site contains information about Informatica, its background, upcoming events, and sales offices. You will also find product and partner information. The services area of the site includes important information about technical support, training and education, and implementation services.

Informatica How-To Library

As an Informatica customer, you can access the Informatica How-To Library at http://mysupport.informatica.com. The How-To Library is a collection of resources to help you learn more about Informatica products and features. It includes articles and interactive demonstrations that provide

solutions to common problems, compare features and behaviors, and guide you through performing specific real-world tasks.

Informatica Knowledge Base

As an Informatica customer, you can access the Informatica Knowledge Base at http://mysupport.informatica.com. Use the Knowledge Base to search for documented solutions to known technical issues about Informatica products. You can also find answers to frequently asked questions, technical white papers, and technical tips. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team through email at KB_Feedback@informatica.com.

Informatica Support YouTube Channel

You can access the Informatica Support YouTube channel at http://www.youtube.com/user/INFASupport. The Informatica Support YouTube channel includes videos about solutions that guide you through performing specific tasks. If you have questions, comments, or ideas about the Informatica Support YouTube channel, contact the Support YouTube team through email at supportvideos@informatica.com or send a tweet to @INFASupport.

Informatica Marketplace

The Informatica Marketplace is a forum where developers and partners can share solutions that augment, extend, or enhance data integration implementations. By leveraging any of the hundreds of solutions available on the Marketplace, you can improve your productivity and speed up time to implementation on your projects. You can access Informatica Marketplace at http://www.informaticamarketplace.com.

Informatica Velocity

You can access Informatica Velocity at http://mysupport.informatica.com. Developed from the real-world experience of hundreds of data management projects, Informatica Velocity represents the collective knowledge of our consultants who have worked with organizations from around the world to plan, develop, deploy, and maintain successful data management solutions. If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Global Customer Support

You can contact a Customer Support Center by telephone or through the Online Support.

Online Support requires a user name and password. You can request a user name and password at http://mysupport.informatica.com.

The telephone numbers for Informatica Global Customer Support are available from the Informatica web site at http://www.informatica.com/us/services-and-training/support-services/global-support-centers/.

CHAPTER 1

Overview

This chapter includes the following topic:

Introduction

Ultra Messaging comprises 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. Ultra Messaging also includes a daemon that implements persistence capabilities. These components provide applications with message delivery functionality that adds considerable value to the basic networking services contained in the host operating system. Applications access Ultra Messaging features through the Ultra Messaging Application Programming Interface (API).

Ultra Messaging includes the following APIs: the *UM C API*, the *UM Java API*, and the *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 the *UM Quick Start Guide* for sample applications in Java and .NET.

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 insure 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.

CHAPTER 2

Fundamental Concepts

This chapter includes the following topics:

Overview

A UM application can function either as a *source* or a *receiver*. A source application sends messages, and a receiver application receives them. (It is also common for an application to function as *both* source and receiver; we separate the concepts for organizational purposes.)

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*, and one or more source programs send to that topic. 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. UM does this through an automatic process known as *topic resolution*.

A topic is just an arbitrary string. For example:

Deals

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. At the present time, messages cannot be *sent* to wildcarded topic names. See "Wildcard Receiver" on page 10.

Persistence

UMP - which contains the Ultra Messaging Streaming Edition (UMS) functionality - includes a component known as the *persistent store*, which provides stable storage (disk or memory) of message streams. UMP delivers a persisted message stream to receiving applications with no additional latency in the vast majority of cases. This offers the functionality of durable subscriptions and confirmed message delivery. Ultra Messaging streaming applications build and run with the UMP persistence feature without modification. See *The Ultra Messaging Guide for Persistence* for more information.

Late Join

In many applications, a new receiver may be interested in messages that were sent before it existed. UM provides a late join feature that allows a new receiver to join a group of others already listening to a source. Without the late join feature, the joining receiver would only receive messages sent after the time it joined. With late join, the source stores sent messages according to its Late Join configuration options so a joining receiver can receive any of these messages that were sent before it joined the group. See "Using Late Join" on page 54.

Request/Response

UM also offers a *Request/Response* messaging model. A sending application (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. UM sends the request message via the normal pub/sub method, whereas UM delivers the response message directly to the requester.

An important aspect of UM's 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 UM 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 UM 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 UM 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 UM receiver objects. And any application can receive and respond to a request, including one that creates and manages UM source objects.

See "Request/Response Model" on page 62.

Transports

A source application uses an *UMS transport* to send messages to a receiver application. A UM transport is built on top of a standard IP protocol. The different UM transport types have different tradeoffs in terms of latency, scalability, throughput, bandwidth sharing, and flexibility. The sending application 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.

A UM sending application can make use of very many topics (over a million). UM maps those topics onto a much smaller number of *transport sessions*. A transport session can be thought of as a specific instance of a transport type. A given transport session might carry a single topic, or might carry hundreds of thousands of topics. A receiving application may express interest in a small set of those topics, in which case UM will join the transport session, receiving messages for *all* topics carried on that transport session. UM will then discard any messages for topics that the application is not interested in. 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.

When UM sets up a transport session and receives the first data over the live data stream, UM generates a BOS (Beginning Of Session) to all receivers that currently exist. When a receiver joins an active transport, this immediately generates a BOS event. When the last topic on a transport session concludes or when a transport path is broken in the network (also referred to as a TCP breakage), UM tears down the transport session and notifies all receivers with an EOS (End Of Session) event. There is no correlation between the deletion of a source by an application and when an EOS is received by a receiver, except if it is the last source sharing the transport.

Note: Non-multicast UM transport types can also use *source-side filtering* to decrease user-space filtering on the receiving side by doing the filtering on the sending side. However, while this might sound attractive at first glance, 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.

See "Transport Objects" on page 12.

Multi-Transport Threads

Part of UM's design is a single threaded model for message data delivery which reduces latency in the receiving CPU. UM, however, also has the ability to distribute data delivery across multiple CPUs by using a receiving thread pool. Receivers created with the configuration option, `use_transport_thread` set to 1 use a thread from the thread pool instead of the context thread. The option, `receive_thread_pool_size` controls the pool size.

As receivers discover new sources through Topic Resolution, UM assigns the network sockets created for the receivers to receive data to either the context thread (default) or to a thread from the pool if `use_transport_thread` is set for the receiver. It is important to understand that thread assignment occurs at the socket level - not the transport level. Transports aggregated on to the same network socket use the same thread.

UM distributes data from different sockets to different threads allowing better process distribution and higher aggregate throughput. Distributing transports across threads also ensures that activity on each transport has no impact on transports assigned to other threads leading to lower latencies in some traffic patterns, e.g. heavy loss conditions.

The following lists restrictions to using multi-transport threads.

- Only *LBT-RM*, *LBT-RU*, *TCP* and *TCP-LB* transport types may be distributed to threads.
- Multi-Transport threads are not supported under `sequential mode` .

- UM processes sources using the same transport socket, e.g. multicast address and port, on the same thread (regardless of the `use_transport_thread` setting. To leverage threading of different sources, assign each source to a different transport destination, e.g. multicast address/port.
- Hot failover sources using LBT-RM on the same topic must not be distributed across threads because they must share the same multicast address and port.
- Hot failover sources using other transport types may not be distributed across threads and must use the context thread.
- Each transport thread has its own Unicast Listener (request) port. Ultra Messaging recommends that you expand the range `request_tcp_port_low` - `request_tcp_port_high` to a larger range when using transport threads. When late join is occurring, UM creates a TCP connection from the transport thread to the source.
- Multi-transport threads are not recommended for use over the UM Router.
- Multi-Transport Threads do not support persistent stores (UMP) or persistent receivers
- Multi-Transport Threads are not compatible with UMDS Server or UMCache

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:

1. A received message on a topic that the application has expressed interest in.
2. 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).
3. 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).
4. New source notification. UM can inform the application when sources are discovered by topic resolution.
5. Receiver loss. UM can inform the application when a data gap is detected that could not be recovered through the normal retransmission mechanism.
6. End of Stream. UM can inform a receiving application when a data stream (transport session) has terminated.

UM delivers events to the application by *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” on page 11.

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.

Rate Controls

For UDP-based transports (LBT-RU and LBT-RM), UM network stability is insured 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.

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. For more information, see the *UM Operations Guide*.

CHAPTER 3

UM Objects

This chapter includes the following topics:

Overview

Many UM documents use the term *object*. Be aware that with the C API, they do *not* refer to formal objects as supported by C++ (i.e. class instances). 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 JNI and C# .NET APIs *are* object oriented, with formal Java/C# objects. See the *Java API documentation* and *.NET API documentation* for more information.

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()**. Your application can give a context a name, which are optional but should be unique across your UM network. You can 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 parameter.
- If you are using plain text UM configuration files, call **lbm_context_attr_setopt()** and specify `context_name` 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`* 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.

Topic Object

A UM *topic* object is conceptually very simple; it is little more than 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 must be bound to source or receiver objects.

A data source creates a topic by calling **lbm_src_topic_alloc()**. A data receiver doesn't explicitly create topic objects; UM does that as topics are discovered and cached. Instead, the receiving application calls **lbm_rcv_topic_lookup()** to find the topic object.

Unlike other objects, the topic object is not created or deleted by the application. UM creates, manages and deletes them internally as needed. However, the application does use them, so the API has functions that give the application access to them when needed (**lbm_src_topic_alloc()** and **lbm_rcv_topic_lookup()**).

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 that must have been previously allocated. A source object can be bound to only one topic. (A topic object, however, can be bound to many sources provided the sources exist in separate contexts.)

Message Properties Object

The message property object allows your application to insert named, typed metadata to topic messages and implement functionality that depends on the message properties. UM allows eight property types: boolean, byte, short, int, long, float, double, and string.

To use message properties, create a message properties object with **lbm_msg_properties_create()**. Then send topic messages with **lbm_src_send_ex()** (or **LBMSource.send()** in the *Java API* or *.NET API*) passing the message properties object through **lbm_src_send_ex_info_t** object. Set the LBM_SRC_SEND_EX_FLAG_PROPERTIES flag on the **lbm_src_send_ex_info_t** object to indicate that it includes properties.

Upon a receipt of a message with properties, your application can access the properties directly through the messages properties field, which is null if no properties are present. Individual property values can be retrieved directly by name, or you can iterate over the collection of properties to determine which properties are present at runtime.

To mitigate any performance impacts in the *C API*, reuse properties objects, **lbm_src_send_ex_info_t** objects and iterators whenever possible. Also limit the number of properties associated with a message. (UM sends the property name and additional indexing information with every message.) In the *Java API* or *.NET API*, also make use of the ZOD feature by calling **Dispose()** on each message before returning from the application callback. This allows property objects to be reused as well.

Note: The Message Properties Object does not support receivers using the arrival order without reassembly setting (option value = 0) of `ordered_delivery`.

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.

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 “Transports” on page 4, 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_transmission_window_size* to 24 MB and the second source sets the same option to 2 MB, UMS assigns 24 MB to the transport session's *transport_lbtrm_transmission_window_size*.

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.

Zero Object Delivery (Source)

The Zero Object Delivery (ZOD) feature for Java and .NET lets sources deliver events to an application with no per-event object creation. (ZOD can also be utilized with context source events.) See "Zero Object Delivery (ZOD)" on page 11 for information on how to employ ZOD.

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 that must have been previously looked up. A receiver object can be bound to only one topic. Multiple receiver objects can be created for the same topic.

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" on page 9, many 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 many 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 the *transport_lbtrm_nak_generation_interval* to 10 seconds and the second receiver sets the same option to 2 seconds, UMS assigns 10 seconds to the transport session's *transport_lbtrm_nak_generation_interval*.

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.

Wildcard Receiver

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 wildcard pattern supplied for matching is a PCRE regular expression that Perl recognizes. See http://perldoc.perl.org/perlrequick.html for details about PCRE. See also the wildcard_receiver pattern_type option in the *UM Configuration Guide*.

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.

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 an example of wildcard usage, see *lbmwrcv.c*

TIBCO™ SmartSockets™ users see the Informatica Knowledge Base article, *Wildcard Topic Regular Expressions*.

Zero Object Delivery (ZOD)

The Zero Object Delivery (ZOD) feature for Java and .NET lets receivers (and sources) deliver messages and events to an application with no per-message or per-event object creation. This facilitates source/receiver applications that would require little to no garbage collection at runtime, producing lower and more consistent message latencies.

To take advantage of this feature, you must call **dispose()** on a message to mark it as available for reuse. To access data from the message when using ZOD, you use a specific pair of LBMMessage-class methods (see below) to extract message data directly from the message, rather than the standard **data()** method. Using the latter method creates a byte array, and consequently, an object. It is the subsequent garbage collecting to recycle those objects that can affect performance.

For using ZOD, the LBMMessage class methods are:

- Java: **dispose()**, **dataBuffer()**, and **dataLength()**
- .NET: **dispose()**, **dataPointer()**, and **length()**

On the other hand, you may need to keep the message as an object for further use after callback. In this case, ZOD is not appropriate and you must call **promote()** on the message, and also you can use **data()** to extract message data.

For more details see the *Java API Overview* or the *.Net LBMMessage Class* description. This feature does not apply to the C API.

Event Queue Object

A UM *event queue* object is conceptually a managed data and control buffer. UM delivers events (including received messages) to your application by means of application callback functions. Without event queues, these callback functions are called from the UM context thread, which places the following restrictions on the application function being called:

1. The application function is not allowed to make certain API calls (mostly having to do with creating or deleting UM objects).
2. The application function must execute very quickly *without* blocking.
3. The application does not have control over when the callback executes. It can't prevent callbacks during critical sections of application code.

Some circumstances require the use of UM event queues. As mentioned above, if the receive callback needs to use UM functions that create or delete objects. Or if the receive callback performs operations that potentially block. You may also want to use an event queue if the receive callback is CPU intensive and can make good use of multiple CPU hardware. Not using an event queue provides the lowest latency, however, high message rates or extensive message processing can negate the low latency benefit if the context thread continually blocks.

Of course, your application can create its own queues, which can be bounded, blocking queues or unbounded, non-blocking queues. For transports that are flow-controlled, a bounded, blocking application queue preserves flow control in your messaging layer because the effect of a filled or blocked queue extends through the message path all the way to source. The speed of the application queue becomes the speed of the source.

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

1. Your application can set a queue size threshold with *queue_size_warning* and be warned when the queue contains too many messages.
2. Your application can set a delay threshold with *queue_delay_warning* and be warned when events have been in the queue for too long.
3. The application callback function has no UM API restrictions.
4. 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.
5. 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()**. In the *Java API* and the *.NET API*, use the `LBMEventQueue` class. An event queue object also holds configuration information that is of *event queue scope*. See *Event Queue Options*.

Transport Objects

This section discusses the following topics.

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:

1. Flow control is desired. For example, when one or more receivers cannot keep up, you wish to slow down the source. This is a "better late than never" philosophy.
2. 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.
3. 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.
4. The application is not sensitive to latency. Use of TCP as a messaging transport can result in unbounded latency.
5. The messages must pass through a restrictive firewall which does not pass multicast traffic.

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 UM Router(s). In the event of a UM Router 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.

You can turn off TCP Session IDs with the UM configuration option, *`transport_tcp_use_session_id`* .

Note: TCP transports may be distributed to receiving threads. See "Multi-Transport Threads" on page 4 for more information.

Transport TCP-LB

The *TCP-LB UMS transport* is a variation on the TCP transport which adds latency-bounded behavior. The source is not flow-controlled as a result of network congestion or slow receivers. So, for applications that require a "better never than late" philosophy, TCP-LB can be a better choice.

However, latency cannot be controlled as tightly as with UDP-based transports (see below). In particular, latency can still be introduced because TCP-LB shares bandwidth equally with other TCP traffic. It also has the same scaling issues as TCP when multiple receivers are present for each topic.

Note: TCP-LB transports may be distributed to receiving threads. See "Multi-Transport Threads" on page 4 for more information.

Transport LBT-RU

The *LBT-RU UMS 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.

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 UM Router(s). In the event of a UM Router 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.

You can turn off LBT-RU Session IDs with the UM configuration option, `transport_lbtru_use_session_id`.

Note: LBT-RU can benefit from hardware acceleration. See *Transport Acceleration Options* in the *UM configuration Guide* for more information.

Note: LBT-RU transports may be distributed to receiving threads. See "Multi-Transport Threads" on page 4 for more information.

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

Note: LBT-RM can benefit from hardware acceleration. See *Transport Acceleration Options* in the *UM configuration Guide* for more information.

Note: LBT-RM transports may be distributed to receiving threads. See "Multi-Transport Threads" on page 4 for more information.

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 can be either source-paced or receiver-paced.

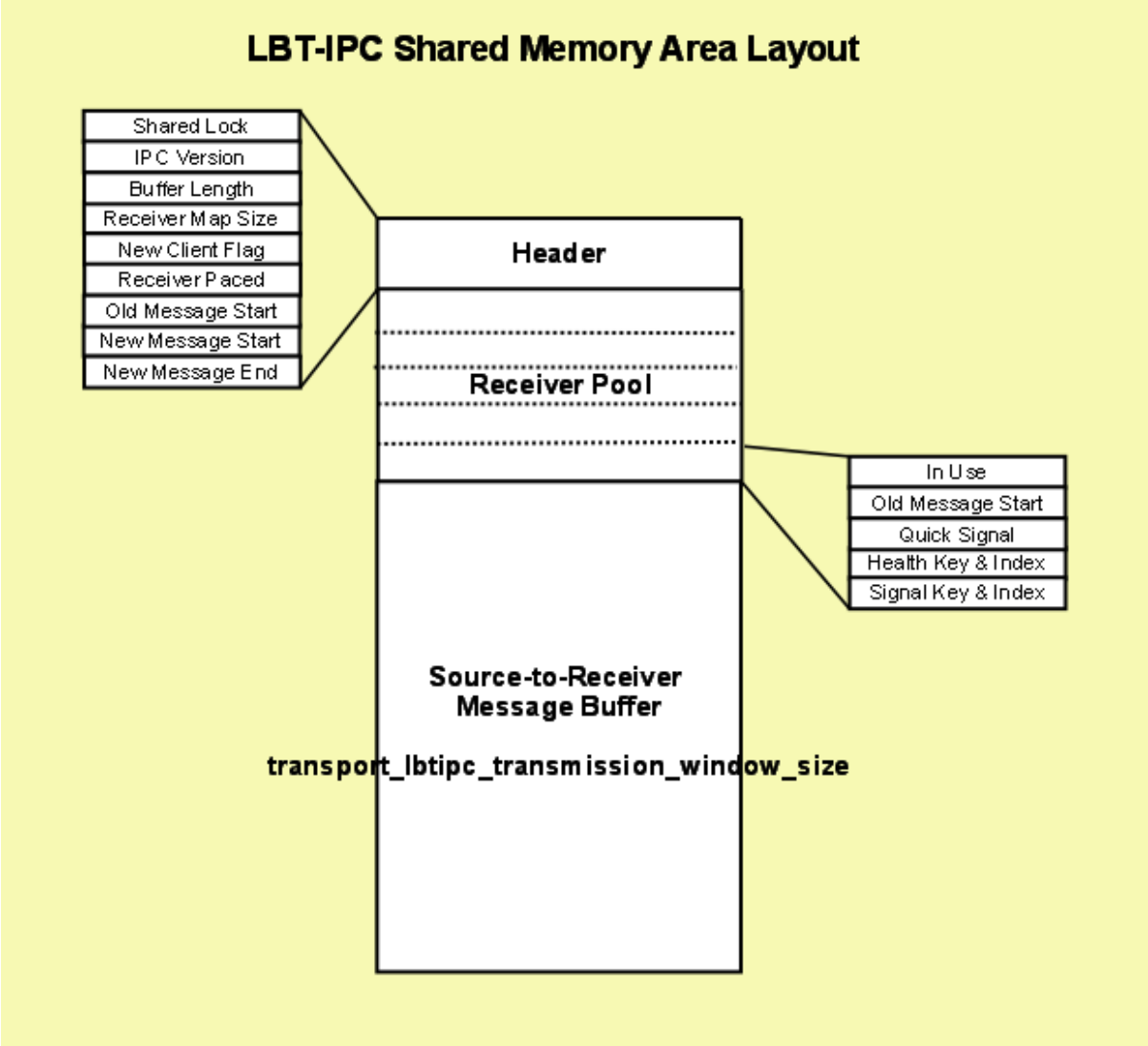
The LBT-IPC transport uses a "lock free" design that eliminates calls to the Operating System and allows receivers quicker access to messages. An internal validation method enacted by receivers while reading messages from the Shared Memory Area ensures message data integrity. The validation method compares 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.

Restriction: Transport LBT-IPC is not supported on the OpenVMS® platform.

LBT-IPC Shared Memory Area

The following diagram illustrates the Shared Memory Area used for LBT-IPC.

Figure 1. LBT-IPC Shared Memory Layout



Header

The Header contains information about the shared memory area resource.

- Shared Lock - shared receiver pool semaphore (mutex on Microsoft Windows) to ensure mutually exclusive access to the receiver pool.
- Version - LBT-IPC version number which is independent of any UM product version number.
- Buffer Length - size of shared memory area.
- Receiver Map Size - Number of entries available in the Receiver Pool which you configure with the source option, *transport_lbtipc_maximum_receivers_per_transport* .
- New Client Flag - set by the receiver after setting its Receiver Pool entry and before releasing the Shared Lock. Indicates to the source that a new receiver has joined the transport.
- Receiver Paced - Indicates if you've configured the transport for receiver-pacing.

- Old Message Start - pointer indicating messages that may be reclaimed.
- New Message Start - pointer indicating messages that may be read.
- New Message End - pointer indicating the end of messages that may be read, which may not be the same as the Old Message Start pointer.

Receiver Pool

The receiver pool is a collection of receiver connections maintained in the Shared Memory Area. The source reads this information if you've configured receiver-pacing to determine if a message can be reclaimed or to monitor a receiver. Each receiver is responsible for finding a free entry in the pool and marking it as used.

- In Use flag - set by receiver while holding the Shared Lock, which effectively indicates the receiver has joined the transport session. Using the Shared Lock ensures mutually exclusive access to the receiver connection pool.
- Oldest Message Start - set by receiver after reading a message. If you enable receiver-pacing the source reads it to determine if message memory can be reclaimed.
- Monitor Shared Lock - checked by the source to monitor a receiver (semaphore on Linux, event on Microsoft Windows).
- Signal Shared Lock - Set by source to notify receiver that new data has been written. (semaphore on Linux, mutex on Microsoft Windows) If you set *`transport_lbtipc_receiver_thread_behavior`* to `busy_wait`, the receiver sets this semaphore to zero and the source does not notify.

Source-to-Receivr Message Buffer

This area contains message data. You specify the size of the shared memory area with a source option, *`transport_lbtipc_transmission_window_size`* . The size of the shared memory area cannot exceed your platform's shared memory area maximum size. UM stores the memory size in the shared memory area's header. The Old Message Start and New Message Start point to positions in this buffer.

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`* and *`transport_lbtipc_id_low`* . You can also specify a shared memory location outside of this range with a source configuration option, *`transport_lbtipc_id`* , 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. Examples names are `LBTIPC_42792ac_20000` or `LBTIPC_66e7c8f6_20001`. Receivers access a shared memory area with this object name to receive (read) topic messages.

Using the configuration option, *`transport_lbtipc_behavior`* , you can choose `source-paced` or `receiver-paced` message transport. See *Transport LBT-IPC Operation Options* in the *UM Configuration Guide*.

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.

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.

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" on page 19.

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-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.

Similarities with Other UM Transports

Although no actual network transport occurs, UM 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.

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.
- Receivers also do not send NAKs when using LBT-IPC.
- LBT-IPC does not support Ordered Delivery options. However, if you set *`ordered_delivery`* 1 or -1 , LBT-IPC reassembles any large messages.
- LBT-IPC does not support Rate Control.

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

Sending to Both Local and Remote Receivers

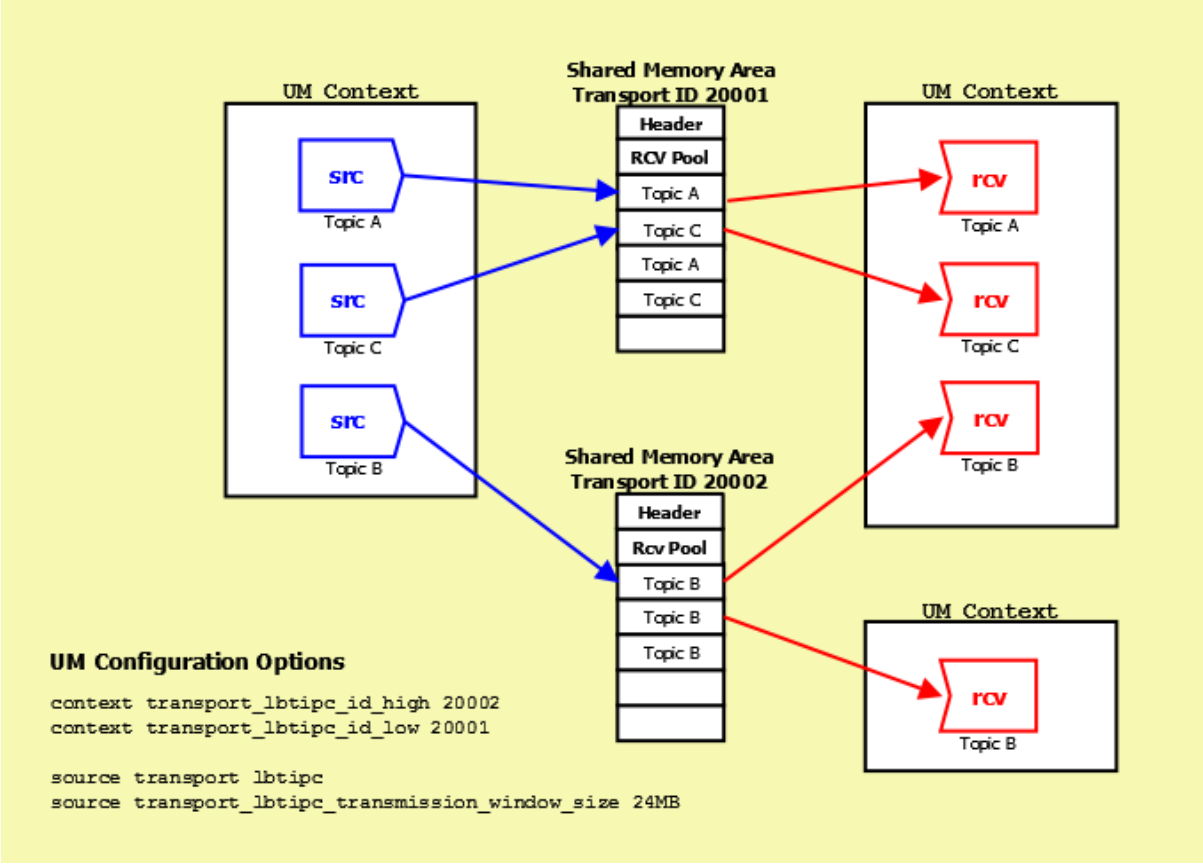
A source application that wants to support both local and remote receivers should create two UM Contexts with different topic resolution configurations, one for IPC sends and one for sends to remote receivers. Separate contexts allows you to use the same topic for both IPC and network sources. If you 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.

LBT-IPC Configuration Example

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

Figure 2. Sending and Receiving with LBT-IPC



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* and *transport_lbtipc_id_low* establish the range of Transport IDs between 20001 and 20002.
- The option *transport_lbtipc* sets the source's transport to LBT-IPC.
- The option *transport_lbtipc_transmission_window_size* sets the size of each Shared Memory Area to 24 MB.

Required Authorities

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.

Host Resource Usage and Limits

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

- Per Source - 1 shared memory segment, 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 segments, 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” on page 20. See also Managing LBT-IPC Host Resources.

LBT-IPC Resource Manager

Deleting an IPC source with `lbm_src_delete()` or deleting an IPC receiver with `lbm_rcv_delete()` 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
```

Discovering Resources in Use

```
$> lbtipc_resource_manager -reclaim

Reclaiming Resources
 Process 24441 still active! Memory resource not reclaimed (SessionID: ab569cec
XPortID: 20001)
 Process 24441 still active! Semaphore resource not reclaimed (Key: 0x68871d75 Sem
Index: 1)
 Process 24436 still active! Semaphore resource not reclaimed (Key: 0x68871d75 Sem
Index: 2)
```

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-SMX is faster than the LBT-IPC transport. Like LBT-IPC, sources can publish topic messages to a shared memory area from which receivers can read topic messages. Unlike LBT-IPC, the native APIs for the LBT-SMX transport are not thread safe and do not support all UM features such as message batching or fragmentation.

You can use either the native LBT-SMX API calls, **lbm_src_buff_acquire()** and **lbm_src_buffs_complete()** to send over LBT-SMX or you can use **lbm_src_send_*()** API calls. The existing send APIs are thread safe with SMX, but they incur a synchronization overhead and thus are slower than the native LBT-SMX API calls.

LBT-SMX operates on the following Ultra Messaging 64-bit packages:

- SunOS-5.10-amd64
- Linux-glibc-2.5-x86_64
- Win2k-x86_64

The example applications, `lbmlatping.*` and `lbmlatpong.*` show how to use the native LBT-SMX API calls. The C API, Java API, and .NET API have identically named example applications. Other example applications can use the LBT-SMX transport with the use of a UM configuration file containing `source transport lbtsmx`. You cannot use LBT-SMX with example applications for features not supported by LBT-SMX, such as `lbmreq.*`, `lbmresp.*`, `lbmrcvq.*` or `lbmwrcvq.*`.

The LBT-SMX configuration options are similar to the LBT-IPC transport options. See *Transport LBT-SMX Operation Options* in the *UM Configuration Guide* for a full explanation of these options.

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, `context transport_lbtsmx_message_statistics_enabled` to `1` to enable the collection of transport statistics.

Sources and LBT-SMX

When you create a source with `lbm_src_create()` and you've set the source's transport configuration 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`* and *`transport_lbtsmx_id_low`* . You can also specify a shared memory location inside or outside of this range with a source configuration option, *`transport_lbtsmx_id`* , to group certain topics in the same shared memory area, if needed. See *Transport LBT-SMX Operation Options* in the *UM Configuration Guide*.

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.

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. Examples names are `LBTSMX_42792ac_20000 or LBTSMX_66e7c8f6_20001`. Receivers access a shared memory area with this object name to receive (read) topic messages.

Sending on a topic with the native LBT-SMX APIs requires the two API calls **lbm_src_buff_acquire()** and **lbm_src_buffs_complete()**. A third convenience API, **lbm_src_buffs_complete_and_acquire()**, combines a call to **lbm_src_buffs_complete()** followed by a call to **lbm_src_buff_acquire()** into one function call to eliminate the overhead of an additional function call.

Important: The native LBT-SMX APIs are not thread safe at the source object or LBT-SMX transport session levels for performance reasons. Applications that use the native API LBT-SMX calls 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.

Note: The native LBT-SMX 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.

Sending over LBT-SMX with Native APIs

Sending with LBT-SMX's native API is a two-step process.

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

   The pointer points directly into the shared memory region. 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. If your application set the LBM_SRC_NONBLOCK flag with **lbm_src_buff_acquire()**, UM returns an LBM_EWOULDBLOCK error condition if the shared memory region does not have enough contiguous space available.

   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`* option minus 16 bytes for headers.

   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 sending application 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 returns another pointer.

Sending over LBT-SMX with Existing APIs

LBT-SMX supports **lbm_src_send_*** API calls. These API calls are fully thread-safe. The LBT-SMX feature restrictions still apply, however, when using **lbm_src_send_*** API calls. 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. See "Differences Between LBT-SMX and Other UM Transports" on page 24

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 **lbm_src_send_*** 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()**.

Attention: Users should not use both the native LBT-SMX APIs and the **lbm_src_send_*** 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()** call succeeds, 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* option minus 16 bytes for headers. Any send API calls with a length parameter greater than this configured value fail.

Receivers and LBT-SMX

Receivers operate identically over LBT-SMX to receivers as all other UM transports. The **msg->data** pointer of a delivered **lbm_msg_t** object points directly into the shared memory region.

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.

Caution: 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.

Similarities Between LBT-SMX and Other UM Transports

Although no actual network transport occurs, UM 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 sending application can send to receivers created by a Java receiving application.

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* 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.
- LBT-SMX does not retransmit messages. Since LBT-SMX transport is a memory write-read operation, messages should not be lost in transit. No retransmission of old messages that have been overwritten occurs.
- Receivers do not send NAKs when using LBT-SMX.

You cannot use the following UM features with LBT-SMX:

- Arrival Order Delivery
- Late Join
- Off Transport Recovery
- Request and Response
- Multi-transport Threads
- Source-side Filtering
- Hot Failover
- Message Properties
- Application Headers
- Implicit and Explicit Message Batching
- Fragmentation and Reassembly
- Immediate Messaging
- Receiver thread behaviors other than "busy_wait"
- Sequential mode receiver threads

You cannot use LBT-SMX to send egress traffic from a UM Dynamic Router to a receiver on a different host or on the same host.

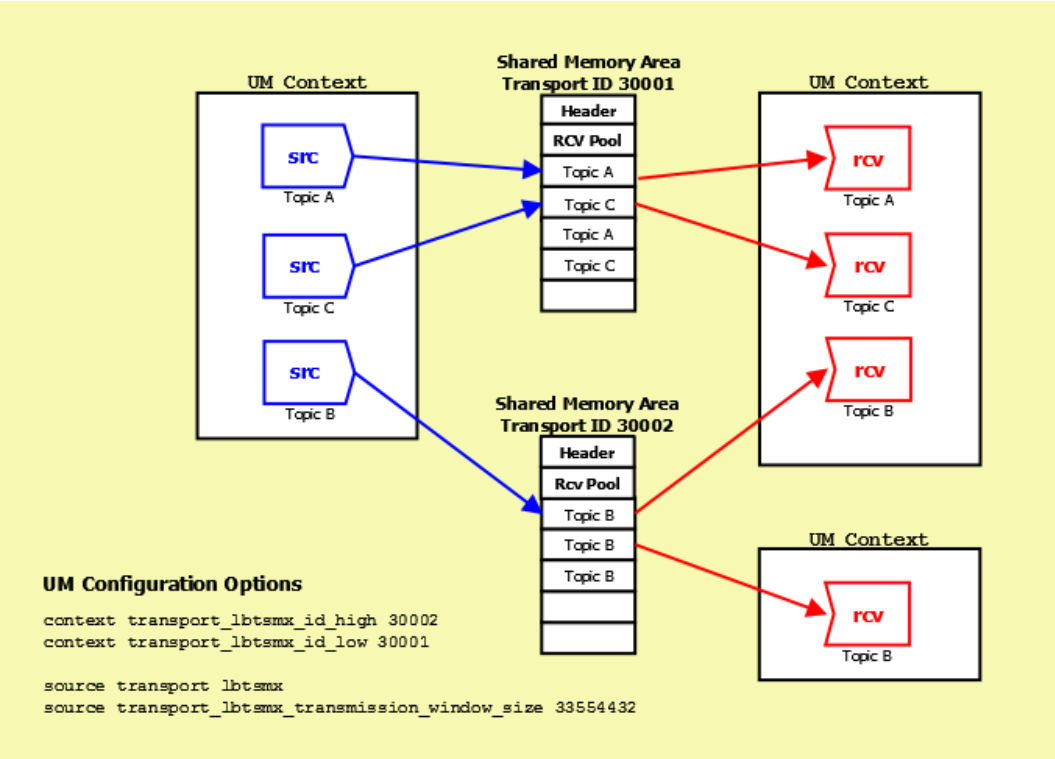
You cannot use LBT-SMX with following UM products:

- Ultra Messaging Persistence Edition
- Ultra Messaging Queuing Edition
- Ultra Messaging Desktop Services
- Ultra Messaging Cache

LBT-SMX Configuration Example

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

Figure 3. Sending and Receiving with LBT-SMX



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 *source transport_lbtsmx_id_high* and *source transport_lbtsmx_id_low* 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 *source transport_lbtsmx_transmission_window_size* 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.

Java Code Examples for LBT-SMX

The Java code examples for LBT-SMX send and receive one million messages. Start the receiver example application before you start the source example application.

Java Source Example

```
import java.nio.ByteBuffer;
import com.latencybusters.lbm.*;

public class SimpleSrc {
    private LBMContext ctx;
    private LBMSource src;

    public static void main(String[] args) {
        try {
            SimpleSrc test = new SimpleSrc();
            test.sendMessages();
            System.out.println("Send Complete");
        } catch (LBMException ex) {
            System.err.println(ex.getMessage());
            ex.printStackTrace();
        }
    }

    public SimpleSrc() throws LBMException
    {
        ctx = new LBMContext();
        LBMSourceAttributes sattr = new LBMSourceAttributes();
        sattr.setValue("transport", "lbtsmx");
        LBMTopic top = ctx.allocTopic("SimpleSmx", sattr);
        src = ctx.createSource(top);
    }

    public void sendMessages() throws LBMException
    {
        /* Keep a reference to the source buffer, which does not change */
        final ByteBuffer srcBuffer = src.getMessagesBuffer();
        /* Sends will block waiting for receivers */
        final int flags = LBM.SRC_BLOCK;
        final int msgLength = 8;
        int pos;

        try { Thread.sleep(1000); } catch (Exception ex) { }
        for (long i = 0; i < 1000000; i++) {
            /* Acquire a position in the buffer */
            pos = src.acquireMessageBufferPosition(msgLength, flags);
            /* Place data at acquired position */
            srcBuffer.putLong(pos, i);
            /* Inform receivers data has been written */
            src.messageBuffersComplete();
        }
        try { Thread.sleep(1000); } catch (Exception ex) { }
        src.close();
        ctx.close();
    }
}
```

The source sends one million messages using the native LBT-SMX Java APIs. **sendMessages()** obtains a reference to the source's message buffer, which does not change for the life of the source. The call **acquireMessageBufferPosition(int, int)** contains the requested message length of 8 bytes. When this call returns, it gives an integer position into the previously obtained messages buffer, which is the position of the message data. UM guarantees that you can safely write the value of the counter `i` into the buffer at this position.

Java Receiver Example

```
import java.nio.ByteBuffer;
import com.latencybusters.lbm.*;
```

```
/* Extend LBMReceiver to avoid onReceive synchronization */
public class SimpleSmxRcv extends LBMReceiver
{
    protected SimpleSmxRcv(LBMContext lbmctx, LBMTopic lbmtopic)
            throws LBMException {
        super(lbmctx, lbmtopic);
    }

    long lastReceivedValue = -1;
    /* Override LBMReceiver onReceive method */
    protected int onReceive(LBMMessage lbmmsg)
    {
        if (lbmmsg.type() == LBM.MSG_DATA) {
            /* New API gets byte buffer with position and limit set */
            ByteBuffer msgsBuffer = lbmmsg.getMessagesBuffer();
            /* Get the message data directly from the buffer */
            lastReceivedValue = msgsBuffer.getLong();
        }
        return 0;
    }

    public static void main(String[] args) {
        LBMContext ctx = null;
        SimpleSmxRcv rcv = null;

        try {
            ctx = new LBMContext();
            LBMTopic top = ctx.lookupTopic("SimpleSmx");
            rcv = new SimpleSmxRcv(ctx, top);
        } catch (LBMException ex) {
            System.out.println(ex.getMessage());
            ex.printStackTrace();
            System.exit(1);
        }

        while (rcv.lastReceivedValue < 999999) {
            try { Thread.sleep(250); } catch (Exception ex) {}
        }
        try {
            rcv.close();
            ctx.close();
            System.out.println("Last Received Value: " + rcv.lastReceivedValue);
        } catch (LBMException ex) {
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

The receiver reads messages from an LBT-SMX Source using the new API on `LBMMessage`. The example extends the `LBMReceiver` class so that you can overwrite the **onReceive()** method, which bypasses synchronization of multiple receiver callbacks. As a result, the **addReceiver()** and **removeReceiver()** methods do not work with this class, but we don't want them anyway. In the overridden **onReceive()** callback, we call **getMessagesBuffer()**, which returns a `ByteBuffer` view of the underlying transport. This allows the application to do *zero copy* reads directly from the memory that stores the message data. The returned `ByteBuffer` position and limit is set to the beginning and end of the message data. The message data does not start at position 0. The application reads a long out of the buffer, which is the same long that was placed by the source example.

Batching

```
public void sendMessages() throws LBMException
{
    ...
    for (long i = 0; i < 1000000; i += 2) {
        /* Acquire a position in the buffer */
        pos = src.acquireMessageBufferPosition(msgLength, flags);
        /* Place data at acquired position */
```

```
            srcBuffer.putLong(pos, i);
            pos = src.acquireMessageBufferPosition(msgLength, flags);
            srcBuffer.putLong(pos, i+1);
            /* Inform receivers two messages have been written */
            src.messageBuffersComplete();
        }
        ...
    }
```

You can implement a batching algorithm at the source by doing multiple acquires before calling complete. When receivers notice that there are new message available, they deliver all new messages in a single loop.

Blocking and Non-blocking Sends

```
public void sendMessages() throws LBMException
{
    ...
    /* Acquire will return -1 if need to wait for receivers */
    final int flags = LBM.SRC_NONBLOCK;
    ...
    for (long i = 0; i < 1000000; i++) {
        /* Acquire a position in the buffer */
        pos = src.acquireMessageBufferPosition(msgLength, flags);
        while (pos == -1) {
            /* Implement a backoff algorithm here */
            try { Thread.sleep(1); } catch (Exception ex) { }
            pos = src.acquireMessageBufferPosition(msgLength, flags);
        }
        /* Place data at acquired position */
        srcBuffer.putLong(pos, i);
        /* Inform receivers data has been written */
        src.messageBuffersComplete();
    }
    ...
}
```

By default, **acquireMessageBufferPosition()** waits for receivers to catch up before it writes the requested number of bytes to the buffer. The resulting *spin wait block* happens only if you did not set the flags argument to LBM.SRC_NONBLOCK. If the flags argument sets the LBM.SRC_NONBLOCK value, then the function returns -1 if the call would have blocked. For performance reasons, **acquireMessageBufferPosition()** does not throw new `LBMEWouldBlock` exceptions like standard send APIs.

Complete and Acquire Function

```
public void sendMessages() throws LBMException
{
    ...
    for (long i = 0; i < 1000000; i++) {
        /* Mark previous acquires complete and reserve space */
        pos = src.messageBuffersCompleteAndAcquirePosition(msgLength, flags);
        /* Place data at acquired position */
        srcBuffer.putLong(pos, i);
    }
    /* final buffers complete after loop */
    src.messageBuffersComplete();
    ...
}
```

The function, **messageBuffersCompleteAndAcquirePosition()**, is a convenience function for the source and calls **messageBuffersComplete()** followed immediately by **acquireMessageBufferPosition()**, which reduces the number of method calls per message.

.NET Code Examples for LBT-SMX

The .NET code examples for LBT-SMX send and receive one million messages. Start the receiver example application before you start the source example application.

.NET Source Example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Runtime.InteropServices;
using com.latencybusters.lbm;

namespace UltraMessagingApplication.SimpleSrc
{
    class SimpleSrc
    {
        LBMContext ctx;
        LBMSource src;

        static void Main(string[] args)
        {
            SimpleSrc test = new SimpleSrc();
            test.sendMessages();
            Console.WriteLine("Send Complete");
        }

        public SimpleSrc()
        {
            ctx = new LBMContext();
            LBMSourceAttributes sattr = new LBMSourceAttributes();
            sattr.setValue("transport", "lbtsmx");
            LBMTopic top = ctx.allocTopic("SimpleSmx", sattr);
            src = ctx.createSource(top);
        }

        private void sendMessages()
        {
            IntPtr writePtr;
            // Sends will block waiting for receivers
            int flags = LBM.SRC_BLOCK;
            uint msgLength = 8;

            Thread.Sleep(1000);

            for (long i = 0; i < 1000000; i++) {
                // Acquire a position in the buffer
                src.buffAcquire(out writePtr, msgLength, flags);
                // Place data at acquired position
                Marshal.WriteInt64(writePtr, i);
                // Inform receivers data has been written
                src.buffsComplete();
            }

            Thread.Sleep(1000);
            src.close();
            ctx.close();
        }
    }
}
```

You can access the shared memory region directly with the `IntPtr` structs. The **src.buffAcquire()** API modifies `writePtr` to point to the next available location in shared memory. When **buffAcquire()** returns, you can safely write to the `writePtr` location up to the length specified in **buffAcquire()**. The **Marshal.WriteInt64()** writes 8 bytes of data to the shared memory region. The call to **buffsComplete()** signals new data to connected receivers.

.NET Receiver Example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Runtime.InteropServices;
using com.latencybusters.lbm;

namespace UltraMessagingApplication.SimpleRcv
{
    class SimpleRcv
    {
        private LBMContext ctx;
        private LBMReceiver rcv;
        private long lastReceivedValue = -1;

        static void Main(string[] args)
        {
            SimpleRcv simpleRcv = new SimpleRcv();
            while (simpleRcv.lastReceivedValue < 999999) {
                Thread.Sleep(250);
            }
            simpleRcv.rcv.close();
            simpleRcv.ctx.close();
            Console.WriteLine("Last Received Value: {0}",
                              simpleRcv.lastReceivedValue);
        }

        public SimpleRcv()
        {
            ctx = new LBMContext();
            LBMTopic top = new LBMTopic(ctx, "SimpleSmx");
            rcv = new LBMReceiver(ctx, top, new LBMReceiverCallback(onReceive), null);
        }

        public int onReceive(Object obj, LBMMessage msg)
        {
            if (msg.type() == LBM.MSG_DATA) {
                // Read data out of shared memory
                lastReceivedValue = Marshal.ReadInt64(msg.dataPointerSafe());
            }
            // dispose the message so the LBMMessage object can be re-used
            msg.dispose();
            return 0;
        }
    }
}
```

The application calls the **simpleRcv::onReceive** callback after the source places new data in the shared memory region. The **msg.dataPointerSafe()** API returns an `IntPtr` to the data, which does not create any new objects. The **Marshal.ReadInt64** API then reads data directly from the shared memory.

Batching

```
private void sendMessages()
{
    ...
    for (int i = 0; i < 1000000; i += 2) {
        // Acquire a position in the buffer
        src.buffAcquire(out writePtr, msgLength, flags);
        // Place data at acquired position
        Marshal.WriteInt32(writePtr, i);
        // Acquire a position in the buffer
        src.buffAcquire(out writePtr, msgLength, flags);
        // Place data at acquired position
        Marshal.WriteInt32(writePtr, i);
        // Inform receivers two messages has been written
        src.buffsComplete();
    }
```

```
    ...
}
```

You can implement a batching algorithm at the source by doing multiple acquires before calling complete. When receivers notice that new message are available, they deliver all new messages in a single loop.

Blocking and Non-blocking Sends

```
private void sendMessages()
{
    ...
    // buffAcquire will return -1 if need to wait for receivers
    int flags = LBM.SRC_NONBLOCK;
    ...
    for (long i = 0; i < 1000000; i++) {
        // Acquire a position in the buffer
        int rc = src.buffAcquire(out writePtr, msgLength, flags);
        while (rc == -1) {
            // Implement a backoff algorithm here
            Thread.Sleep(0);
            rc = src.buffAcquire(out writePtr, msgLength, flags);
        }
        // Place data at acquired position
        Marshal.WriteInt64(writePtr, i);
        // Inform receivers that a message has been written
        src.buffsComplete();
    }
    ...
}
```

By default, **buffAcquire()** waits for receivers to catch up before it writes the requested number of bytes to the buffer. The resulting *spin wait block* happens only if you did not set the flags argument to LBM.SRC_NONBLOCK. If the flags argument sets the LBM.SRC_NONBLOCK value, then the function returns -1 if the call would have blocked. For performance reasons, **buffAcquire()** does not throw new `LBMEWouldBlock` exceptions like standard send APIs.

Complete and Acquire Function

```
private void sendMessages()
{
    ...
    for (long i = 0; i < 1000000; i++) {
        // Acquire a position in the buffer
        src.buffsCompleteAndAcquire(out writePtr, msgLength, flags);
        // Place data at acquired position
        Marshal.WriteInt64(writePtr, i);
    }

    // final buffsComplete after loop
    src.buffsComplete();

    ...
}
```

The function, **buffsCompleteAndAcquire()**, is a convenience function for the source and calls **buffsComplete()** followed immediately by **buffAcquire()**, which reduces the number of method calls per message.

Reduce Synchronization Overhead

```
public SimpleRcv()
{
    ctx = new LBMContext();
    LBMReceiverAttributes rattr = new LBMReceiverAttributes();
    // Set the enableSingleReceiverCallback attribute to 'true'
    rattr.enableSingleReceiverCallback(true);
    LBMTopic top = new LBMTopic(ctx, "SimpleSmx", rattr);
    // With enableSingleReceiverCallback, a callback must be specified in the ver
constructor.
```

```
        rcv = new LBMReceiver(ctx, top, new LBMReceiverCallback(onReceive), null);
        // rcv.addReceiver and rcv.removeReceiver will result in log warnings.
    }
```

Delivery latency to an `LBMReceiver` callback can be reduced with a single callback. Call **LBMReceiverAttributes::enableSingleReceiverCallback** on the attributes object used to create the `LBMReceiver`. The **addReceiver()** and **removeReceiver()** APIs become defunct, and your application calls the application receiver callback without any locks taken. The **enableSingelReceiverCallback()** API eliminates callback related synchronization overhead.

Note: In Java, inheriting from `LBMReceiver` and overriding the **onReceive** can achieve the same thing.

Increase Performance with unsafe Code Constructs

```
for (long i = 0; i < 1000000; i++) {
    // Acquire a position in the buffer
    src.buffAcquire(out writePtr, msgLength, flags);
    // Place data at acquired position
    unsafe {
        *((long*)(writePtr)) = i;
    }
    // Inform receivers data has been written
    src.buffsComplete();
}

public int onReceive(Object obj, LBMMessage msg)
{
    if (msg.type() == LBM.MSG_DATA) {
        unsafe {
            lastReceivedValue = *((long*)msg.dataPointer());
        }
    }
    // dispose the message so the object can be re-used
    msg.dispose();
    return 0;
}
```

Using .NET unsafe code constructs can increase performance. By manipulating pointers directly, you can eliminate calls to external APIs, resulting in lower latencies.

Transport LBT-RDMA

The LBT-RDMA transport is Remote Direct Memory Access (RDMA) UM transport that allows sources to publish topic messages to a shared memory area from which receivers can read topic messages. LBT-RDMA runs across InfiniBand and 10 Gigabit Ethernet hardware.

Note: Use of the LBT-RDMA transport requires the purchase and installation of the Ultra Messaging RDMA Transport Module. See your Ultra Messaging representative for licensing specifics.

Restriction: Transport LBT-RDMA is supported on only the X86 Linux 64-bit platform.

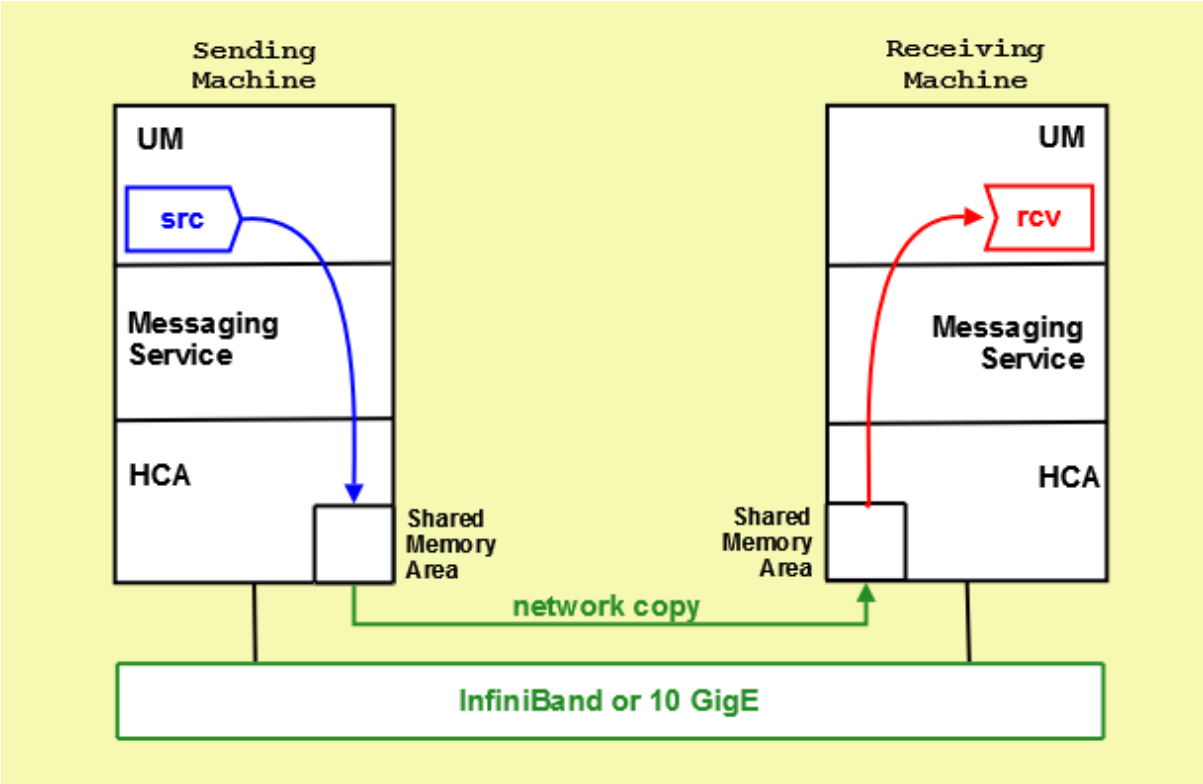
When you create a source with `lbm_src_create()` and you've set the transport option to RDMA, UM creates a shared memory area object on the sending machine's Host Channel Adapter (HCA) card. UM assigns one of the RDMA transport ports to this area specified with the UM context configuration options, *`transport_lbtrdma_port_high`* and *`transport_lbtrdma_port_low`* . You can also specify a shared memory location outside of this range with a source configuration option, *`transport_lbtrdma_port`* , to prioritize certain topics, if needed.

When you create a receiver with `lbm_rcv_create()` for a topic being sent over LBT-RDMA, UM creates a shared memory area on the receiving machine's HCA card. The network hardware immediately copies any new data from the sending HCA to the receiving HCA. UM receivers monitor the receiving shared memory area for new topic messages. You configure receiver monitoring with *`transport_lbtrdma_receiver_thread_behavior`* .

LBT-RDMA Object Diagram

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

Figure 4. Sending and Receiving with LBT-RDMA



Similarities with Other UMS Transports

UM functions in much the same way as if you send packets across a traditional Ethernet network as with other UM transports.

- If you use a range of ports, UM assigns multiple topics that have been sent by multiple sources in a round robin manner to all the transport sessions configured my the port range.
- Transport sessions assume the configuration option values of the first source assigned to the transport session.
- Sources are subject to message batching.
- Topic resolution operates identically with LBT-RDMA as other UM transports albeit with a new advertisement type, `LBMRDMA`.

Differences from Other UMS Transports

- Unlike LBT-RM which uses a transmission window to specify a buffer size to retain messages in case they must be retransmitted, LBT-RDMA uses the transmission window option to establish the size of the shared memory.
- LBT-RDMA does not retransmit messages. Since LBT-RDMA transport is essentially a memory write/read operation, messages should not be lost in transit. However, if the shared memory area fills up, new

messages overwrite old messages, leading to unrecoverable transport loss. Note that a persisted stream with UMP or UMQ can provide off-transport recovery for unrecoverable transport loss. No retransmission of old messages that have been overwritten occurs.

- Receivers also do not send NAKs when using LBT-RDMA.
- LBT-RDMA is inherently ordered in its message delivery. If you set *ordered_delivery* to 0, then UM delivers message fragments individually in sequence number order, without reassembly.
- LBT-RDMA is source-paced but does not support Rate Control. If the source message rate exceeds the receiver's consumption rate, unrecoverable message loss eventually occurs.
- LBT-RDMA creates a separate receiver thread in the receiving context.

CHAPTER 4

Architecture

This chapter includes the following topics:

Overview

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.

Embedded Mode

When you create a context (**lbm_context_create()**) with the UM configuration option *operational_mode* 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.

Sequential Mode

When you create a context (**lbm_context_create()**) with the UM configuration option *operational_mode* set to `sequential`, the context thread is NOT created. It becomes the application's responsibility to donate a thread to UM by calling **lbm_context_process_events()** regularly, typically in a tight loop. Use Sequential mode for circumstances where your application wants control over the attributes of the context thread. For

example, some applications raise the priority of the context thread so as to obtain more consistent latencies. In sequential mode, no separate thread is spawned when a context is created.

You enable Sequential mode with the following configuration option.

```
context operational_mode sequential
```

Topic Resolution

Topic resolution is the discovery of a topic's transport session information by a receiver to enable the receipt of topic messages. By default, UM relies on multicast requests and responses to resolve topics to transport sessions. (You can also use Unicast requests and responses, if needed.) UM receivers multicast their topic requests, or queries, to an IP multicast address and UDP port configured with the UM configuration options, *`resolver_multicast_address`* and *`resolver_multicast_port`*). UM sources also multicast their advertisements and responses to receiver queries to the same multicast address and UDP port.

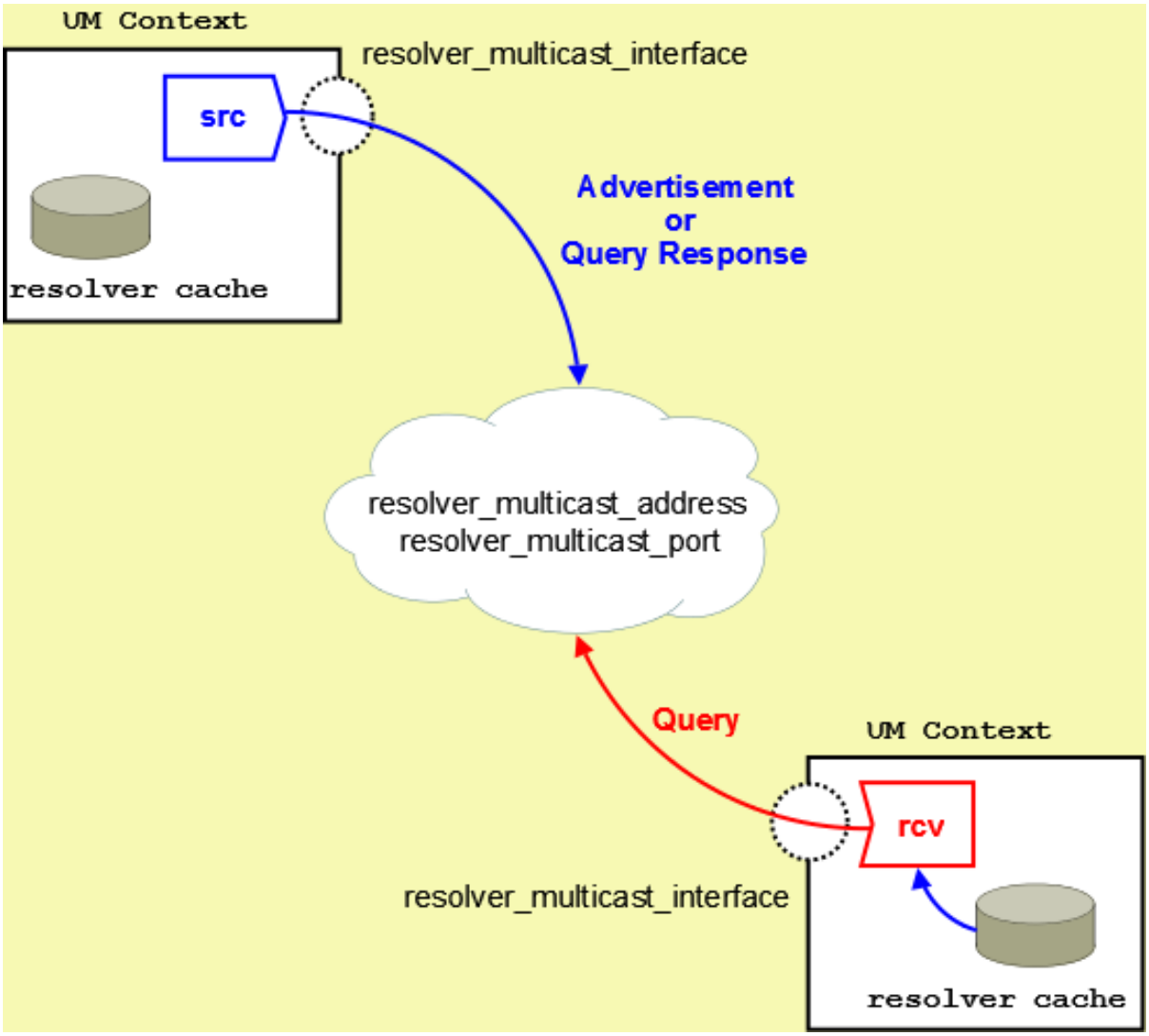
Topic Resolution occurs in the following phases:

- **Initial Phase** - Period that allows you to resolve a topic aggressively. Can be used to resolve all known topics before message sending begins. 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 "steady state" period during which a topic is resolved and UM uses no system resources for topic resolution.

Multicast Topic Resolution

The following diagram depicts the UM topic resolution using multicast.

Figure 5. Topic Resolution via Multicast



UM performs topic resolution automatically. Your application does not need to call any API functions to initiate topic resolution, however, you can influence topic resolution with "Topic Resolution Configuration Options" on page 43. Moreover, you can set configuration options for individual topics by using the **lbm_*_attr_setopt()** functions in your application. See "Assigning Different Configuration Options to Individual Topics" on page 44

Topic Resolution also occurs across UM Routers, which means between Topic Resolution Domains. A Topic Resolution Domain refers to all the UM contexts that use the same UM topic resolution configuration values, such as `resolver_multicast_address` and `resolver_multicast_port` . UM Routers automatically assign Topic Resolution Domain IDs and manage Topic resolution traffic across them. See the *UM Router Guide* for more information.

Note: Multicast topic resolution traffic can benefit from hardware acceleration. See *Transport Acceleration Options* in the *UM Configuration Guide* for more information.

Restriction: Multicast Topic Resolution is not directly supported on the OpenVMS® platform. UM applications running on the OpenVMS® platform, however, can use multicast topic resolution running on a different platform, such as Microsoft Windows® or Linux.

Sources Advertise

UM sources help UM receivers discover transport information in the following ways.

- Advertise Active Topics - Each source advertises its active topic first upon its creation and subsequently according to the *resolver_advertisement_*_interval* configuration options for the Initial and Sustaining Phases. Sources advertise by sending a Topic Information Record (TIR). (You can prevent a source from sending an advertisement upon creation with *resolver_send_initial_advertisement* .)
- Respond to Topic Queries - Each source responds immediately to queries from receivers about its topic.

Both a topic advertisement and a query response contain the topic's transport session information. Based on the transport type, a receiver can join the appropriate multicast group (for LBT-RM), send a connection request (for LBT-RU), connect to the source (for TCP) or access a shared memory area (for LBT-IPC). The address and port information potentially contained within a TIR includes:

- For a TCP transport, the source address, TCP port and Session ID.
- For an LBT-RM transport, the unicast UDP port (to which NAKs are sent) and the UDP destination port.
- For an LBT-RU transport, the source address, UDP port and Session ID.
- For an LBT-IPC transport, the Host ID, LBT-IPC Session ID and Transport ID.

  For an LBT-RDMA transport, the source address, RDMA port and Session ID.

See *Resolver Operation Options* in the *UM Configuration Guide* for more information.

Note: Any sources you configure to send to UMP stores by setting the *ume_store* configuration option, also include a UMP flag in their advertisements. This indicates that the receiver should request the source to send a Source Registration Information (SRI) record, which identifies the store or stores the receiver should register with. See *Source and Receiver Registration with the Store* in the *UM Guide for Persistence* for more information.

Receivers Query

Receivers can discover transport information in the following ways.

- Search advertisements collected in the resolver cache maintained by the UM context.
- Listen for source advertisements on the `resolver_multicast_address:port`.
- Send a topic query (TQR).

A new receiver queries for its topic according to the *resolver_query_*_interval* configuration options for the Initial and Sustaining Phases.

Note: The *resolver_query_minimum_initial_interval* actually begins after you call prior to creating the receiver. If you have disabled the Initial Phase for the topic's resolution, the *resolver_query_sustaining_interval* begins after you call **lbm_rcv_topic_lookup()**.

A Topic Query Record (TQR) consists primarily of the topic string. Receivers continue querying on a topic until they discover the number of sources configured by *resolution_number_of_sources_query_threshold* . However the large default of this configuration option (10,000,000) allows a receiver to continue to query until both the initial and sustaining phase of topic resolution complete.

See *Resolver Operation Options* in the *UM Configuration Guide* for more information.

Wildcard Receivers

Wildcard receivers can discover transport information in the following ways.

- Search advertisements collected in the resolver cache maintained by the UM context.
- Listen for source advertisements on the `resolver_multicast_address:port`.
- Send a wildcard receiver topic query (WC-TQR).

UM implements only one phase of wildcard receiver queries, sending wildcard receiver queries according to wildcard receiver *`resolver_query_*_interval`* configuration options until the topic pattern has been queried for the *`resolver_query_minimum_duration`* . The wildcard receiver topic query (WC-TQR) contains the topic pattern and the *`pattern_type`* .

See *Wildcard Receiver Options* in the *UM Configuration Guide* for more information.

Topic Resolution Phases

The phases of topic resolution pertain to individual topics. Therefore if your system has 100 topics, 100 different topic resolution advertisement and query phases may be running concurrently. This describes the three phases of Ultra Messaging topic resolution.

- "Initial Phase" on page 39
- "Sustaining Phase" on page 41
- "Quiescent Phase" on page 43

Initial Phase

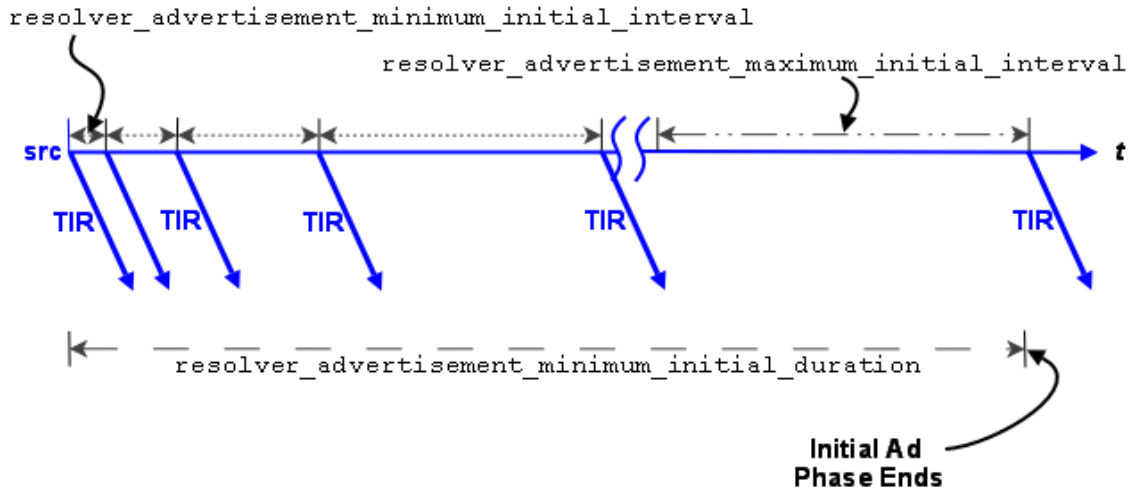
The initial topic resolution phase for a topic is an aggressive phase that can be used to resolve all topics before sending any messages. During the initial phase, network traffic and CPU utilization might actually be higher. You can completely disable this phase, if desired. See *Disabling Aspects of Topic Resolution* in the *UM Configuration Guide*.

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`* . 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`* . 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 . This concludes the initial advertisement phase for the topic.

Figure 6. Initial Advertisement Phase



The initial phase for a topic can take longer than the *resolver_advertisement_minimum_initial_duration* if many topics are in resolution at the same time. The configuration options, *resolver_initial_advertisements_per_second* and *resolver_initial_advertisement_bps* 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.

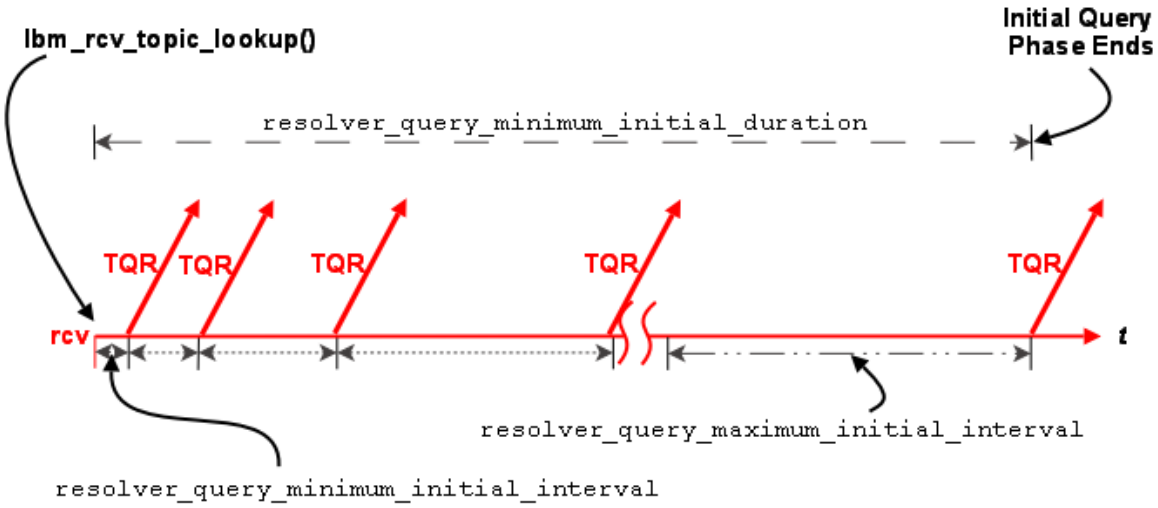
If a source advertising in the initial phase receives a topic query, it responds with a topic advertisement. UM recalculates the next advertisement interval from that point forward as if the advertisement was sent at the nearest interval.

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* default is 20 ms. Subsequent intervals double in length until the interval reaches 200 ms, or the

resolver_query_maximum_initial_interval . The query interval remains at 200 ms until the initial querying phase reaches 5000 ms, or the *resolver_query_minimum_initial_duration* .

Figure 7. Initial Query Phase



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

Sustaining Phase

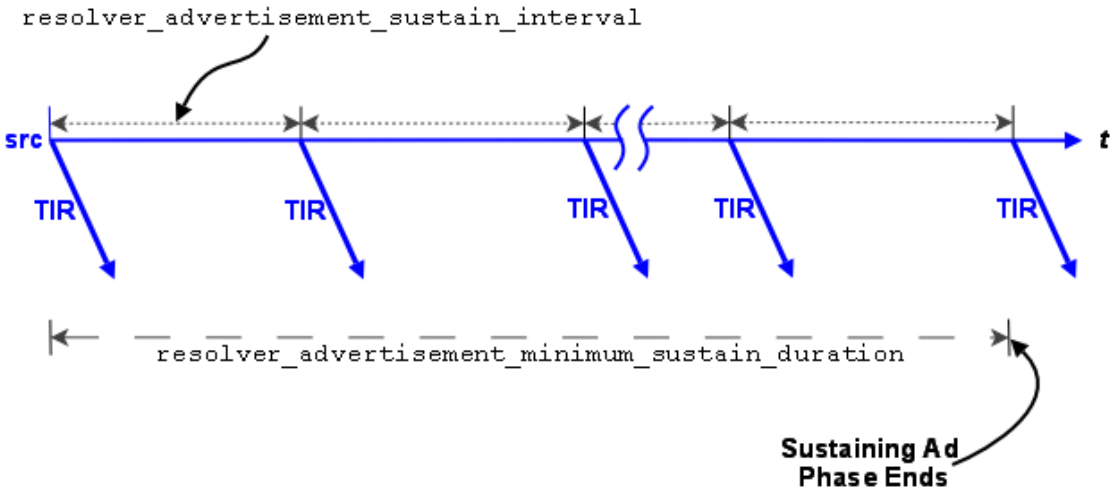
The sustaining topic resolution phase follows the initial phase and can be a less active phase in which a new receiver resolves its topic. It can also act as the sole topic resolution phase if you disable the initial phase. The sustaining phase defaults use less network resources than the initial phase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

Advertising in the Sustaining Phase

For the sustaining phase defaults, a source sends an advertisement every second (*resolver_advertisement_sustain_interval*) for 1 minute (*resolver_advertisement_minimum_sustain_duration*). 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.

Figure 8. Sustaining Advertisement Phase

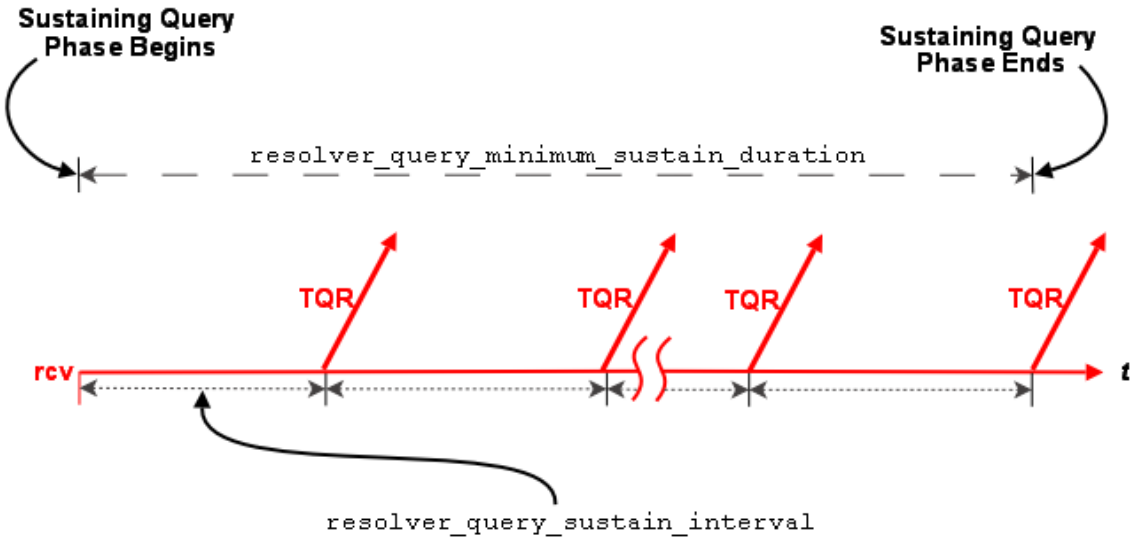


The sustaining advertisement phase has UM context-wide rate limit controls (*resolver_sustain_advertisements_per_second* and *resolver_sustain_advertisement_bps*) that can result in the extension of a phase's duration in the case of a large number of topics or long topic names.

Querying in the Sustaining Phase

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

Figure 9. Sustaining Query Phase



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

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

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

Quiescent Phase

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.

Store (context) Name Resolution

Topic resolution facilitates the resolution of 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* in the *UM Guide for Persistence* for more information.

If you do not use UM Routers, the `DomainID` is zero. Otherwise, the `DomainID` represents the Topic Resolution Domain where the store resides. Stores can learn their `DomainID` by listening to Topic Resolution traffic. See the *UM Router Guide* for more information about Topic Resolution Domains.

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 a store names (*ume_store_name*), 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* . 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* . 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* . 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`.

Topic Resolution Configuration Options

Refer to the UM Configuration Guide for specific information about Topic Resolution Configuration Options.

- *Resolver Operation Options*

- *Multicast Resolver Network Options*
- *Unicast Resolver Network Options*
- *Wildcard Receiver Options*

Assigning Different Configuration Options to Individual Topics

You can assign different configuration option values to individual topics by accessing the topic attribute table (`lbm_*_topic_attr_t_stct`) before creating the source, receiver or wildcard receiver.

Creating a Source with Different Topic Resolution Options

1. Call `lbm_src_topic_attr_setopt()` to set new option value
2. Call `lbm_src_topic_alloc()`
3. Call `lbm_src_create()`

Creating a Receiver with Different Topic Resolution Options

1. Call `lbm_rcv_topic_attr_setopt()` to set new option value
2. Call `lbm_rcv_topic_lookup()`
3. Call `lbm_rcv_create()`

Creating a Wildcard Receiver with Different Topic Resolution Options

1. Call `lbm_wildcard_rcv_attr_setopt()` to set new wildcard receiver option value
2. Call `lbm_wildcard_rcv_create()`

Multicast Network Options

Essentially, the `_incoming` and `_outgoing` versions of `resolver_multicast_address/port` provide more fine-grained control of topic resolution. By default, the *`resolver_multicast_address`* and *`resolver_multicast_port`* and the `_incoming` and `_outgoing` address and port are set to the same value. If you want your context to listen to a particular multicast address/port and send on another address/port, then you can set the `_incoming` and `_outgoing` configuration options to different values.

See *Resolver Operation Options* in the *UM Configuration Guide* for more information.

Unicast 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 UM unicast topic resolution daemon(s) (Chapter 6, “Manpage for lbmrd” on page 89), which perform the same topic resolution activities as multicast topic resolution. You configure each instance of the unicast topic resolution daemon with *`resolver_unicast_daemon`* .

The `lbmrd` can run on any machine, including the source or receiver (enter `lbmrd` -h for instructions). 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`* . This registration with the `lbmrd` allows the client to receive advertisements or queries from `lbmrd`. `lbmrd` maintains no state about topics, only about clients.

Restriction: Unicast Topic Resolution is not supported on the OpenVMS® platform. UM applications running on the OpenVMS® platform, however, can use unicast topic resolution running on a different platform, such as Microsoft Windows® or Linux.

LBMRD with the UM Router Best Practice

If you're using the `lbmrd` for topic resolution across UM Routers, 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`* , 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 tradeoff is a little more network traffic every 5 seconds.

Unicast Topic Information Records

Of all topic resolution messages, only the TIR contains address and port information. This tells a receiver how it can get the data being published. Based on the transport type, a receiver can join the appropriate multicast group (for LBT-RM), send a connection request (for LBT-RU), or connect to the source (for TCP).

The address and port information potentially contained within a TIR includes:

- For a TCP transport, the source address and TCP port.
- For an LBT-RM transport, the unicast UDP port (to which NAKs are sent) and the UDP destination port.
- For an LBT-RU transport, the source address and UDP port.

For unicast-based transports (TCP and LBT-RU), the TIR source address is 0.0.0.0, not the actual source address.

Topic resolution messages (whether received by the application via multicast, or by the unicast topic resolution daemon via unicast) are always UDP datagrams. They are received via a `recvfrom()` call, which also obtains the address and port from which the datagrams were received. If the address 0.0.0.0 (INADDR_ANY) appears for one of the addresses, `lbmrd` replaces it with the address from which the datagram is received. The net effect is as if the actual source address had originally been put into the TIR.

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.

lbmrd Configuration File

This section presents the syntax of the `lbmrd` configuration file, which is an XML file. Descriptions of elements also appear below. See also *Unicast Resolver Example Configuration* in the *UM Configuration Guide* for an example `lbmrd` configuration file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
```

```
    <domains>
     <domain name="domain-name-1">
         <network>network-specification</network>
     </domain>
     <domain name="domain-name-2">
         <network>network-specification</network>
     </domain>
    </domains>
    <transformations>
     <transform source="source-domain-name"
         destination="destination-domain-name">
         <rule>
          <match address="original-address" port="original-port"/>
          <replace address="replacement-address" port="replacement-port"/>
         </rule>
     </transform>
    </transformations>
</lbmrd>
```

<lbmrd> Element

The <lbmrd> element is the root element. It requires a single attribute, version, which defines the version of the DTD to be used. Currently, only version 1.0 is supported. The <lbmrd> element must contain a single <domains> element and a single <transformations> element.

<domains> Element

The <domains> element defines the set of network domains. The <domains> element may contain one or more <domain> elements. Each defines a separate domain.

<domain> Element Element

The <domain> element defines a single network domain. Each domain must be named via the name attribute. This name is referenced in <map> elements, which are discussed below. Each domain name must be unique. The <domain> element may contain one or more <network> elements.

<network> Element Element

The <network> element defines a single network specification which is to be considered part of the enclosing <domain>. The network specification must contain either an IP address, or a network specification in CIDR form.

<transformations> Element

The <transformations> element defines and contains the set of transformations to be applied to the TIRs. The <transformations> element contains one or more <transform> elements, described below.

<transformation> Element

The <transform> element 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 a network domain name as defined by the <domain> elements. The <transform> element contains one or more <rule> elements, described below.

<rule> Element

Each <rule> element is associated with the enclosing <transform> element, and completes the transformation tuple. The <rule> element must contain one <match> element, and one <replace> element, described below.

<match> Element

The <match> element defines the address and port to match within the TIR. The attributes address and port specify the address and port. address must specify a full IP address (a network specification is not permitted). port specifies the port in the TIR. To match any port, specify port="*" (which is the default).

<replace> Element

The <replace> element defines the address and port which are to replace those matched in the TIR. The attributes address and port specify the address and port. address must specify a full IP address (a network specification is not permitted). To leave the TIR port unchanged, specify port="*" (which is the default).

It is valid to specify port="*" for both <match> and <replace>. This effectively matches all ports for the given address and changes only the address. It is important to note that TIR addresses and ports are considered together. For example, the Ultra Messaging R for the Enterprise option in the TIR contains the source address and port, and the store address and port. When processing a transformation tuple, the source address and source port are considered (and transformed) together, and the store address and store port are considered (and transformed) together.

Unicast Topic Resolution Across Administrative Domains

If your network architecture includes remote or local LANs that use Network Address Translation (NAT), you can implement an `lbmrd` configuration file to translate IP addresses/ports across administrative domains. Without translation, `lbmrd` clients (sources and receivers) across NAT boundaries cannot connect to each other in response to topic advertisements due to NAT restrictions.

By default, topic advertisements forwarded by `lbmrd` contain the private (or inside) address/port of the source. Routers implementing NAT prevent connection to these private addresses from receivers outside the LAN.

The `lbmrd` configuration file allows `lbmrd` to insert a translation or outside address/port for the private address/port of the source in the topic advertisement. This outside or translation address must already be configured in the router's static NAT table. When the receiver attempts to connect to the source by using the source address/port in the topic advertisement, the NAT router automatically translates the outside address/port to the private address/port, thereby allowing the connection.

Note: The Request/Response model and the Late Join feature work only with (`lbmrd`) across local LANs that use Network Address Translation (NAT) if you use the default value (0.0.0.0) for *`request_tcp_interface`*.

See the *UM Configuration Guide* for more information.

Message 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. However, the implementation of batching is almost exclusively the realm of the sender.

Many people are under the impression that while batching improves CPU load, it increases message latency. While it is true that there are circumstances where this can happen, it is also true that careful use of batching can result in small latency increases or none at all. In fact, there are circumstances where batching can actually reduce latency.

UM allows the following methods for batching messages.

- “Implicit Batching” on page 48 - the default behavior, batching messages for individual transport sessions.

- "Adaptive Batching" on page 49 - a convenience feature of UM that monitors sending activity and automatically determines the optimum time to flush the Implicit Batch buffer.
- "Intelligent Batching" on page 49 - a method that makes use of your application's knowledge of the messages it must send, clearing the Implicit Batching buffer when sending the only remaining message.
- "Explicit Batching" on page 50 - provides greater control to your application through `lbm_src_send()` message flags and also operates in conjunction with the Implicit Batching mechanism.
- "Application Batching" on page 50 - your application groups messages and sends them in a single batch.

Implicit Batching

UM automatically batches smaller messages into transport session datagrams. The implicit batching configuration options, *`implicit_batching_interval`* (default = 200 milliseconds) and *`implicit_batching_minimum_length`* (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* in the *UM Configuration Guide*.

See also "Source Configuration and Transport Sessions" on page 9.

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. See "Explicit Batching" on page 50 for information about topic-level message batching.

Implicit Batching Operation

Implicit Batching buffers messages until:

- the buffer size exceeds the configured *`implicit_batching_minimum_length`* or
- the oldest message in the buffer has been in the buffer for *`implicit_batching_interval`* milliseconds.

When either condition is met, UM flushes the buffer, pushing the messages onto the network.

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`* 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` to push out the data (i.e. if the entire transport session has periods of inactivity longer than the value of *`implicit_batching_interval`* (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" on page 49.

The minimum value for the *`implicit_batching_interval`* 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 approximately 20 milliseconds. On Microsoft Windows machines, the time is probably 16 milliseconds. On a Linux 2.6 kernel, the actual time is 3 milliseconds. Using a `implicit_batching_interval` value of 3 guarantees the minimum possible wait for whichever operating system you are using.

Implicit Batching Example

The following example demonstrates how the `implicit_batching_minimum_length` is actually a trigger or floor, for sending batched messages. It is sometimes misconstrued as a ceiling or upper limit.

```
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 Operating System fragments the datagram into packets independently of UM and reassembles them on the receiving end.
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 tradeoff 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" on page 49 which can provide low latency and high throughput with a single setting.

Adaptive Batching

Adaptive Batching is a convenience batching feature that attempts to send messages immediately during periods of low volume and automatically batch messages during periods of higher volume. The goal of Adaptive Batching is to automatically optimize throughput and latency by monitoring such things as the time between calls to **lbm_src_send()**, the time messages spend in the Implicit Batching queue, the Rate Controller queue, and other sending activities. With this information, Adaptive Batching determines if sending batched messages now or later produces the least latency.

Adaptive Batching will not satisfy everyone's requirements of throughput and latency. You only need to turn it on and determine if it produces satisfactory performance. If it does, you need do nothing more. If you are not satisfied with the results, simply turn it off.

You enable Adaptive Batching by setting *`implicit_batching_type`* to `adaptive`. When using Adaptive Batching, it is advisable to increase the *`implicit_batching_minimum_length`* option to a higher value.

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 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 UMS 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* 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.

Explicit Batching

UM allows you to batch messages for a particular topic with explicit batching. 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" on page 48 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* 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* and your application has a batch of messages to send that exceeds the `implicit_batching_minimum_length`. 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.

```
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` but is below the 8192 maximum datagram size.
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` is again exceeded, or the `implicit_batching_interval` is met, or a message arrives in the buffer with the flush flag (LBM_MSG_FLUSH) set.

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.

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 Not Reassembled
- Arrival Order, Fragments Reassembled

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` configuration option as shown:

```
receiver ordered_delivery 1
```

Please note that ordered delivery can introduce latency when packets are lost.

Arrival Order, Fragments Not Reassembled

This mode allows messages to be delivered to the application in the order they 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 guarantees the lowest latency.

With this mode, the receiver delivers messages larger than the transport's maximum datagram size as individual fragments. (See transport_*_datagram_max_size in the Ultra Messaging Configuration Guide.) 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, Fragments Reassembled

This mode delivers messages in the order they are received, except for fragmented messages, which UM reassembles before delivering to your application. Your application can then use the sequence_number field of lbm_msg_t objects to order or discard messages.

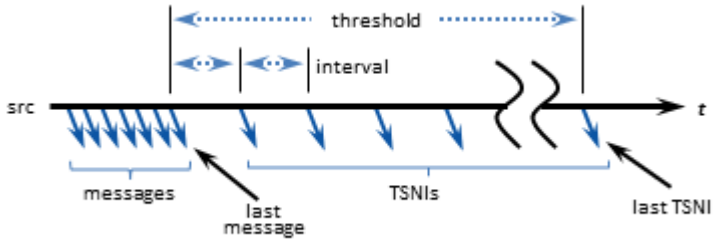
To enable this arrival-order-with-reassembly mode, set the following configuration option as shown:

```
receiver ordered_delivery -1
```

Loss Detection Using TSNIs

When a source enters a period during which it has no data traffic to send, that source issues timed Topic Sequence Number Info (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 between TSNIs.

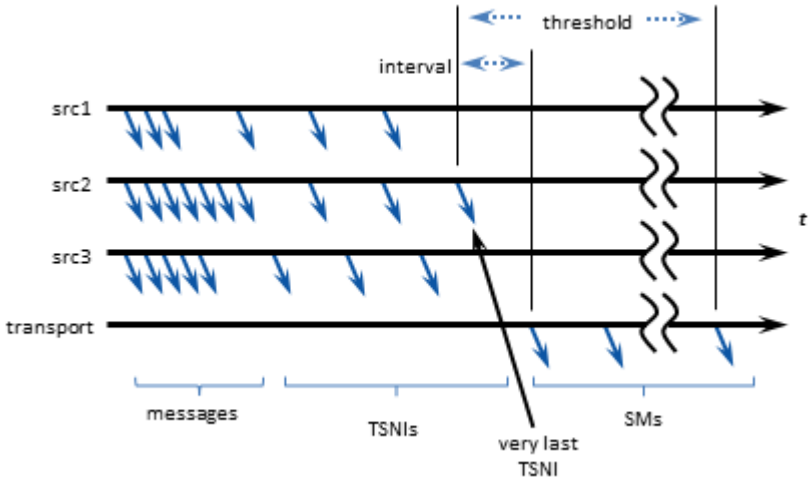
Sources send TSNIs over the same transport and on the same topic as normal data messages. You can set a time value of the TSNI interval with configuration option `transport_topic_sequence_number_info_interval`. You can also set a time value for the duration that the source sends contiguous TSNIs with configuration option `transport_topic_sequence_number_info_active_threshold`, after which time the source stops issuing TSNIs.



Receiver Keepalive Using Sesssion Messages

When an LBT-RM, LBT-RU, or LBT-IPC transport session enters a period during which it has no data traffic to send, UM issues timed Session Messages (SMs). For example, suppose all topics in a session stop sending data. One by one, they then send TSNIs, and if there is still no data to send, their TSNI periods eventually expire. After the last quiescent topic's TSNIs stop, UM begins transmitting SMs.

You can set time values for SM interval and duration with configuration options specific to their transport type.


threshold
interval
src1
src2
src3
transport
t
messages
TSNIs
very last
TSNI
SMs

CHAPTER 5

UMS Features

This chapter includes the following topics:

Using Late Join

This section introduces the use of UM Late Join in default and specialized configurations. Specifically, this section on UM Late Join includes:

See the *UM Configuration Guide* for specific information about Late Join configuration options.

Note: If your application is running within a UM context with configuration option `request_tcp_bind_request_port` set to zero, then request port binding has been turned off, which also disables the Late Join feature.

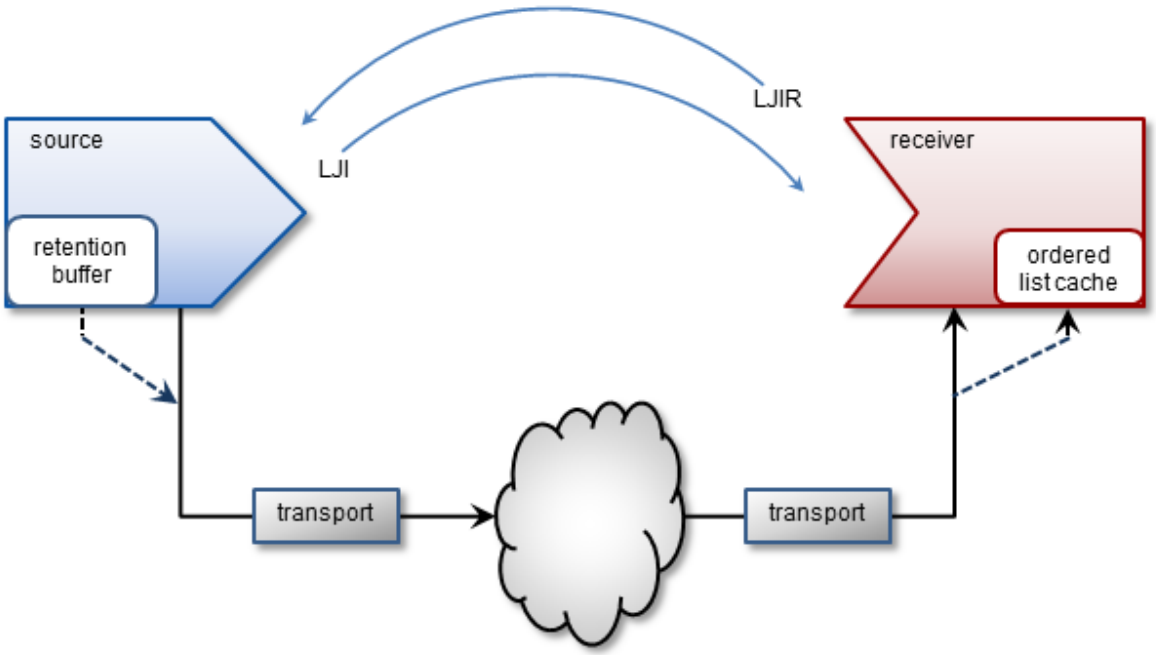
Late Join Overview

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* completes topic resolution. Topic advertisements from the source contain a flag that indicates the source is configured for Late Join with *late_join*.
2. The new receiver sends a Late Join Initiation Request (LJIR) to request a previously transmitted messages. The receiver configuration option, *retransmit_request_outstanding_maximum*, 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" on page 59, the receiver issues additional requests, and the source retransmits these additional groups of older messages, oldest first.

Figure 10. Late Join Message Path



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*, the source deletes older messages as it adds new ones. The source configuration option *retransmit_retention_age_threshold*, controls message deletion based on message age.

Note: 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.

Attention: If you set the receiver configuration option *ordered_delivery* 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*, 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.

Late Join With UMP

Late Join can be implemented in conjunction with UMP's persistent store feature, however in this configuration, it functions somewhat differently. 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.

Late Join Options Summary

Following is a summary of Late join configuration options. Please refer to *UM Configuration Guide* for full descriptions of these options.

scope (object)	option
source	*late_join*
source	*retransmit_retention_age_threshold*
source	*retransmit_retention_size_limit*
source	*retransmit_retention_size_threshold*
receiver	*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*

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” on page 58.)

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* 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* 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.

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* 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 retransmit_request_maximum 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.)

Configuring Late Join for Large Numbers of Messages

Suppose you have a 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.

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* is set. set. For example, when set to the default of 200, the receiver sends requests the first 200 messages (Sequence Number 0 - 199). Upon receiving Sequence Number 0, it then requests the next message (200), and so on, limiting the number of outstanding unfulfilled requests to 200.

Note that in some environments, the default of 200 messages may be too high and overwhelm receivers with RXs, which can cause loss in a live LBT-RM stream. However, in other situations higher values can increase the rate of RXs received.

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 figure 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* 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 is 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.

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 UM Router. This section describes the OTR feature.

OTR Overview

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 UMP 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 source `late_join` 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`, the `otr_request_initial_delay` 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` and `otr_request_maximum_interval` to set the initial (minimum) and maximum intervals, respectively.

The source retransmits lost messages to the recovered receiver via unicast.

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` and `retransmit_message_caching_proximity`.

The option `otr_message_caching_threshold` 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`, which is also used by Late Join (see "retransmit_message_caching_proximity (receiver)" on page 59), 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.

OTR With UMP

You can implement OTR in conjunction with UMP's persistent store feature, however in this configuration, it functions somewhat differently. 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`* 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.

OTR Options Summary

The following set of configuration options govern OTR functionality. Please refer to the *Ultra Messaging Configuration Guide* for full descriptions of these options. You can click the individual links below for each option's description.

scope (object)	option
source	*late_join*
source	*retransmit_retention_age_threshold*
source	*retransmit_retention_size_limit*
source	*retransmit_retention_size_threshold*
receiver	*use_otr*
receiver	*otr_request_message_timeout*
receiver	*otr_request_initial_delay*
receiver	*otr_request_log_alert_cooldown*
receiver	*otr_request_maximum_interval*

scope (object)	option
receiver	*otr_request_minimum_interval*
receiver	*otr_request_outstanding_maximum*
receiver	*otr_message_caching_threshold*
receiver	*retransmit_message_caching_proximity*

Request/Response Model

This section discusses the following topics.

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" on page 79.
- **lbm_unicast_immediate_request()** to send a request to a topic as a unicast immediate message. See "Multicast Immediate Messaging" on page 79.

The request function returns a request object and defines an application callback for responses that allows the receiving application to send a response directly to the requesting application via a special TCP connection instead of a normal data transport. The requesting application -- not UM -- determines how many responses it needs. Therefore, it must delete the request object when it no longer wants to receive responses by calling **lbm_request_delete()**. It discards any responses that arrive after the request object has been deleted.

Response Message

An application responds to an UM request message by calling **lbm_send_response()**. Contained within that request message's header is a response object, which serves as a return address to the requester. UM passes the response object to **lbm_send_response()**. Since the response object is part of the message header, it is deleted at the same time that the message is deleted. Therefore, if the sending of the response cannot be done within the responder's receive callback, the message must be retained and subsequently deleted.

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`, 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` 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.

Configuration

See the UM Configuration Guide for the descriptions of the Request/Response configuration options.

- *Request Network Options*
- *Request Operations Options*
- *Response Operation Options*

Note: If your application is running within an UM context where the configuration option, *`request_tcp_bind_request_port`* has been set to zero, request port binding has been turned off, which also disables the Request/Response feature.

Example Applications

UM includes two example applications that illustrate Request/Response.

- *lbmreq.c* - application that sends requests on a given topic (single source) and waits for responses. See also the Java example, *lbmreq.java*, and the .NET example, *lbmreq.cs*.
- *lbmresp.c* - application that waits for requests and sends responses back on a given topic (single receiver). See also the Java example, *lbmresp.java*, and the .NET example, *lbmresp.cs*.

We can demonstrate a series of 5 requests and responses with the following procedure.

1. Run `lbmresp -v topicname`
2. 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
```

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 Endianess 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.

Use the following links to access a complete reference of SDM functions, field types and message field operations.

- *C Application Programmer's Interface* — click on the Files tab at the top and select `lbmsdm.h`.
- *Java Application Programmer's Interface* — select `com.latencybusters.lbm.sdm` under Packages.
- *.NET Application Programmer's Interface* — select the `com.latencybusters.lbm.sdm` Namespace.

Restriction: The Self-Describing Messaging (SDM) feature is not supported on the OpenVMS® platform.

Pre-Defined Messaging

The UM Pre-Defined Messaging (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 than 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 Application Programmer's Interfaces for complete references of PDM functions, field types and message field operations. 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 the following API documentation:

- *C Application Programmer's Interface* — click on the Files tab at the top and select `lbmpdm.h`.
- *Java Application Programmer's Interface* — select `com.latencybusters.lbm.pdm` under Packages.
- *.NET Application Programmer's Interface* — select the `com.latencybusters.lbm.pdm` Namespace.

Restriction: The Pre-Defined Messaging (PDM) feature is not supported on the OpenVMS® platform.

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.

Set up a source to do the following:

1. Add field values to the message instance.
2. Serialize the message so that it can be sent.

Likewise, set up a receiver to do the following:

1. Deserialize the received bytes into the message instance.

2. Extract the field values from the message.

Getting Started

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.

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);
        }
    }
}
```

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);
    }
}
```

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 **sourceCreateMessageWithPDM** 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 **LBMSDMFieldInt8**) 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 **LBMSDMessage**.

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**, **PDMTimestamp**, 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.DECIMAL**, the following code would be used to set the value.

```
PDMDecimal decimal = new PDMDecimal((long)2, (byte)32);
srcPDMMsg.setFieldValue(fldInfo103, decimal);
```

Multicast Immediate Messaging

As an alternative to the normal, source-based UM messaging model, Multicast Immediate Messaging (MIM) offers advantages to short-lived topics and applications that cannot tolerate a delay between source creation and the sending of the first message. See the Knowledge Base article, *Avoiding or Minimizing Delay Before Sending* for background on this delay and other head-loss mitigation techniques.

Multicast Immediate Messaging avoids delay by eliminating the topic resolution process. MIM accomplishes this by:

1. Configuring transport information into sending and receiving applications.
2. Including topic strings within each message.

MIM is well-suited to applications where a small number of messages are sent to a topic. By eliminating topic resolution, MIM also reduces one of the causes of head-loss, defined as the loss of initial messages sent over a new transport session. Messages sent before topic resolution is complete will be lost.

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).

Note: Multicast Immediate Messaging can benefit from hardware acceleration. See *Transport Acceleration Options* in the *UM Configuration Guide* for more information.

Temporary Transport Session

MIM uses the same reliable multicast algorithms as LBT-RM. When a sending application 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` 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 sending applications use a long deletion timeout if they continue to use MIM after significant periods of inactivity.

MIM forces all topics across all sending applications 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.

MIM Notifications

MIM notifications differ in the following ways from normal UM source-based sending.

- When a sending application's MIM transport session times out and is deleted, the receiving applications do not receive an EOS notification.
- Applications with a source notification callback are not informed of a MIM sender. Since source notification is basically a hook into the topic resolution system, this should not come as a surprise.
- 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 “Loss Handling” on page 81.

Receiving Immediate Messages

MIM does not require any special type of receiver. It uses the topic-based publish/subscribe model so an application must still create a receiver for a topic to receive MIM messages.

Note: If needed, an application can send topic-less 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 non-topic immediate messages.

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.

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.

MIM Configuration

As of UM 3.1, MIM supports ordered delivery. As of UM 3.3.2, the MIM configuration option, *`mim_ordered_delivery`* defaults to ordered delivery.

See the UM Configuration Guide for the descriptions of the MIM configuration options.

- *`Multicast Immediate Messaging Network Options`*
- *`Multicast Immediate Messaging Reliability Options`*
- *`Multicast Immediate Messaging Operation Options`*

Note: Setting *`mim_incoming_address`* to 0.0.0.0 turns off MIM.

MIM Example Applications

UM includes two example applications that illustrate MIM.

- *lbmimsg.c* - application that sends immediate messages as fast as it can to a given topic (single source). See also the Java example, *lbmimsg.java* and the .NET example, *lbmimsg.cs*.
- *lbmireq.c* - application that sends immediate requests to a given topic (single source) and waits for responses.

lbmimsg.c

We can demonstrate the default operation of Immediate Messaging with lbmimsg and lbmrcv.

1. Run `lbmrcv -v topicName`
2. Run `lbmimsg topicName`

The lbmrcv output should resemble the following.

```
Immediate messaging target: TCP:10.29.1.78:14391
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][0], 25 bytes
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][1], 25 bytes
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][2], 25 bytes
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][3], 25 bytes
```

```
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][4], 25 bytes
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][5], 25 bytes
[topicName][LBTRM:10.29.1.78:14390:644c8862:224.10.10.21:14401][6], 25 bytes
```

Each line in the lbmrcv output is a message received, showing the topic name, transport type, receiver IP:Port, multicast address and message number.

lbmireq.c

Sending an UM request by MIM can be demonstrated with lbmireq and lbmrcv, which shows a single request being sent by lbmireq and received by lbmrcv. (lbmrcv sends no response.)

1. Run `lbmrcv -v topicName`
2. Run `lbmireq topicName`

lbmrcv

The lbmrcv output should resemble the following.

```
$ lbmrcv -v topicName
Immediate messaging target: TCP:10.29.1.78:14391
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
[topicName][LBTRM:10.29.1.78:14390:92100885:224.10.10.21:14401][0], Request
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
1     secs.  0     Kmsgs/sec.  0     Kbps
```

lbmireq

The lbmireq output should resemble the following.

```
$ lbmireq topicName
Using TCP port 4392 for responses
Sending 1 requests of size 25 bytes to target <> topic <topicName>
Sending request 0
Sent request 0. Pausing 5 seconds.
Done waiting for responses. 0 responses (0 bytes) received. Deleting request
Quitting...
Lingering for 5 seconds...
```

Spectrum

UM Spectrum, which refers to a "spectrum of channels", allows a source application to allocate any number of channels using **lbm_src_channel_create()** on which to send (**lbm_src_send_ex()**) different messages of the same topic. A receiving application can subscribe receivers to one or more channels with either `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.

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.

See also the *C API* documentation.

Performance Pluses

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.

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.

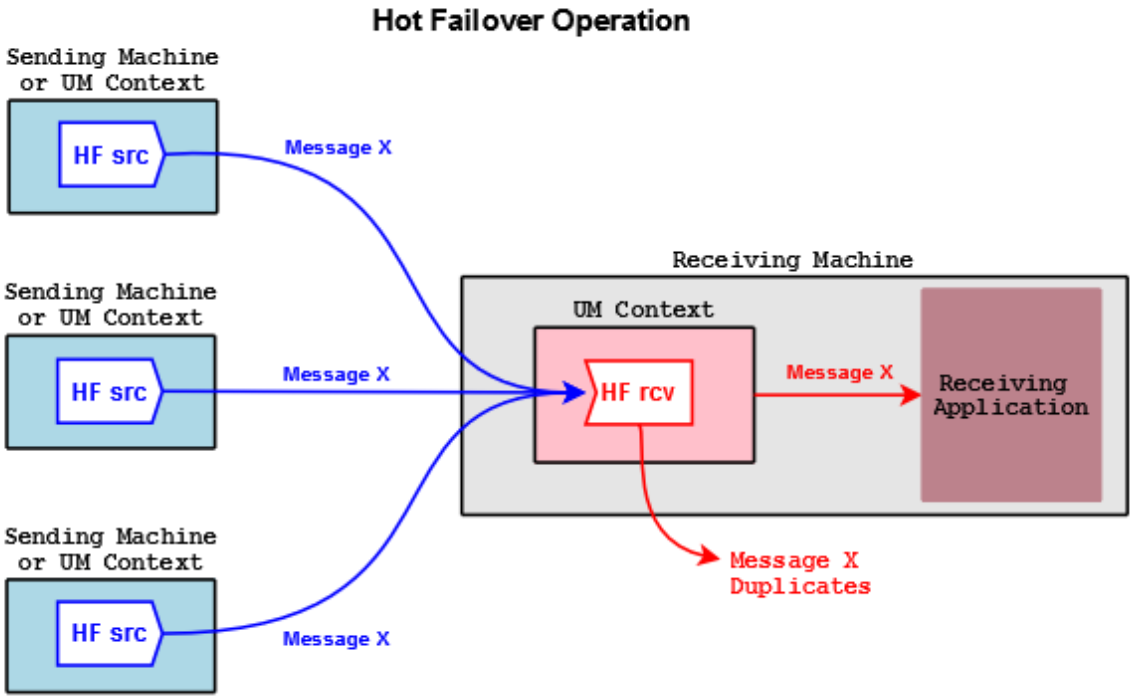
- `null_channel_behavior` - behavior for messages delivered with no channel information.
- `unrecognized_channel_behavior` - behavior for messages delivered with channel information but are on a channel for which the receiver has not registered interest.
- `channel_map_tablesz` - controls the size of the table used by a receiver to store channel subscriptions.

Hot Failover

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.

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.

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 is 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 lbmhfsrc in the `UM Examples Page`.

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`, `transport_lbtru_nak_generation_interval`). 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 `lbmhfrcv` in the *UM Examples Page*.

Implementing Hot Failover Wildcard Receivers

To create an HF wildcard receiver, set option `hf_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`.

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.

Using Hot Failover with UMP

When implementing Hot Failover with UMP, you must consider the following impact on hardware resources:

- Additional storage space required for a UMP disk store
- Higher disk activity
- Higher network activity
- Increased application complexity regarding message filtering

Also note that you must enable UME explicit ACKs and Hot Failover duplicate delivery in each Hot Failover receiving application.

For detailed information on using Hot Failover with UMP, see the Knowledge Base article *FAQ: Is UMP compatible with Hot Failover?* .

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.

```
HF source 1 sends message sequence numbers: 10, 11, 12, 13, 25, 26, 38
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
```

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*). 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.

```
HF source 1 sends message sequence numbers: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o,
19o, 20
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
```

Using Hot Failover with Ordered Delivery

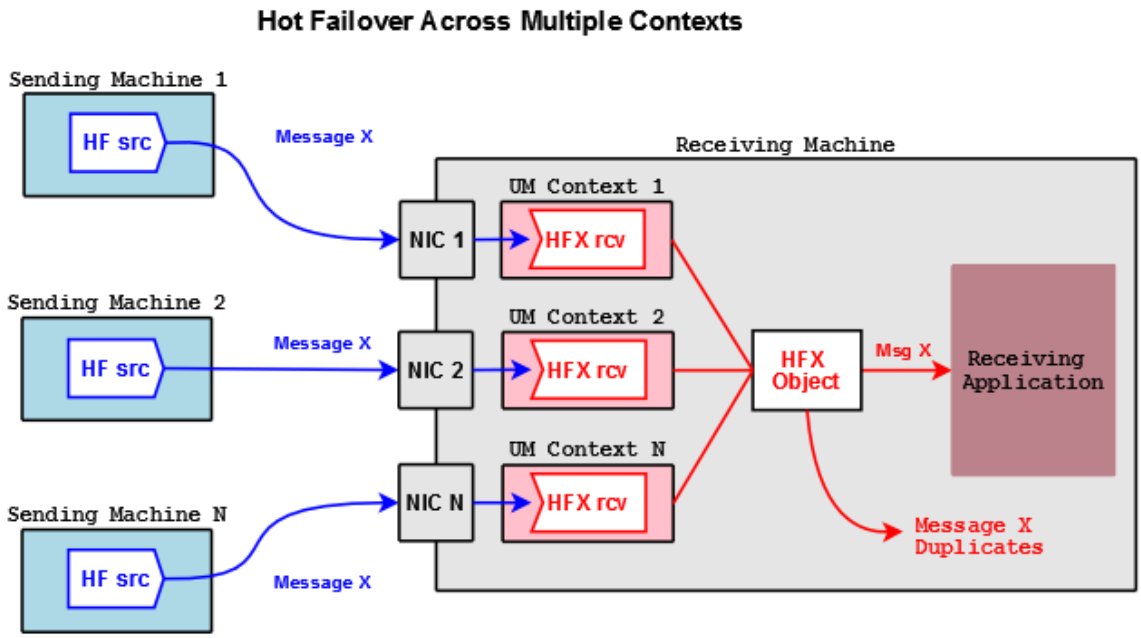
An HF receiver takes some of its operating parameters directly from the receive topic attributes. The `ordered_delivery` setting indicates the ordering for the HF receiver. Please see "Ordered Delivery" on page 51 for information on the different modes of delivery order.

Note: UM supports Arrival Order with HF only when all sources use the same transport type.

Hot Failover Across Multiple Contexts

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()** 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 also ...

- *Hot Failover Operation Options* for HFX Configuration Options.

- `LBMHFX*.java` in *UM Java API*.
- `LBMHFX*.cs` in *UM .NET API*.

CHAPTER 6

Manpage for lbmrd

This chapter includes the following topic:

lbmrd

lbmrd [-a] [--activity] [-d] [--dump-dtd] [-h] [--help] [-i] [--interface] [-L] [--logfile] [-p] [--port] [-t] [--ttl] [-v] [--validate] *config-file*

Description

Resolver services for UM messaging products are provided by lbmrd.

The -i and -p (or --interface and --port) options identify the network interface IP address and port that lbmrd opens to listen for unicast topic resolution traffic. The defaults are `INADDR_ANY` and `15380`, respectively.

The -a and -t (or --activity and --ttl) options interact to detect and remove "dead" clients, i.e., UMS/UME 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.

Option -t 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.

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.

NOTE: 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 .

The -r option sets the receive socket buffer size in bytes .

The output is written to a log file if either -L or --logfile is supplied.

The DTD used to validate a configuration file will be dumped to standard output with the -d or --dump-dtd option. After dumping the DTD, lbmrd exits immediately.

The configuration file will be validated against the DTD if either the -v or --validate options are given. After attempting validation, lbmrd exits immediately. The exit status will be 0 for a configuration file validated by the DTD and non-zero otherwise.

Command line help is available with -h or --help.

Exit Status

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