Informatica

Ultra Messaging (Version 6.12)

Concepts Guide

Contents

Chapter 1

Introduction

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

Attention

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

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. The UMP and UMQ products also include a "store" daemon that implements Persistence. The UMQ product also includes a "broker" daemon that implements Brokered Queuing.

Applications access Ultra Messaging features through the Ultra Messaging Application Programming Interface (A↩ PI).

Ultra Messaging includes the following APIs: the UM C API, the UM Java API, and the UM .NET API. 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 UMQ product also supports the JMS API.

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.

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

Chapter 2

Fundamental Concepts

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.)

2.1 Topic Structure and Management

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

A topic is just an arbitrary string. For example:

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

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

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

2.1.1 Message Ordering

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

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

- The topics are from a single publisher,
- The topics are mapped to the same Transport Session,

- The Transport Session is configured for TCP, IPC, or SMX,
- The subscriber is in the same Topic Resolution Domain (TRD) as the publisher (no DRO in the data path),
- The messages being received are "live" - i.e. not being recovered from late join, OTR, or Persistence,
- The subscriber is not participating in queuing,
- The subscriber is not using Hot Failover.

2.1.2 Topic Resolution Overview

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

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

For more information, see Topic Resolution Description.

2.1.3 Topic Resolution Domain

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

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

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

For more information, see Topic Resolution Description.

2.2 Persistence

The UMP and UMQ products include a component known as the Persistent Store, which provides stable storage (disk or memory) of message streams. UM 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 Persistence feature without modification. For more information, see the `UM Guide for Persistence`.

2.3 Queuing

The UMQ product, which contains Streaming and Persistence functionality, also includes message queuing capabilities. See `UM Guide to Queuing` for more information.

2.4 UM Router

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

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

See The `Dynamic Routing Guide` for more information.

2.5 Late Join

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

Source-side configuration options:

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

Receiver-side configuration options:

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

Note

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

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

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

2.6 Request/Response

Ultra Messaging 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. Ultra Messaging sends the request message via the normal pub/sub method, whereas Ultra Messaging delivers the response message directly to the requester.

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

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

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

Note

You cannot use Request/Response with Queuing functionality (UMQ).

2.7 UM Transports

A source application uses a UM transport to send messages to a receiver application. An Ultra Messaging transport type is built on top of a standard IP protocol. For example, the UM transport type "LBT-RM" is built on top of the standard UDP protocol using standard multicast addressing. The different Ultra Messaging transport types have different trade offs in terms of latency, scalability, throughput, bandwidth sharing, and flexibility. The 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.

2.7.1 Transport Sessions

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

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

Note

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

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

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

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

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

If the publisher deletes a topic source, the subscribing application may or may not get an immediate EOS event (End Of Stream), depending on different circumstances. For example, in many cases, the deletion of topic sources by a publisher will not trigger an EOS event until *all* sources mapped to a Transport Session are deleted. When the last topic is deleted, the Transport Session itself is deleted, and an EOS event might then be delivered to *all* topic receivers that were mapped to that Transport Session. Note that for UDP transports, the deletion of a Transport Session by the publisher is not immediately detected by a subscriber, until an activity timeout expires.

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

Note

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

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

2.7.2 Multi-Transport Threads

Warning

> The "Multi-Transport Threads" (MTT) feature is deprecated as of UM version 6.9 and will be eliminated from a future UM version. It is replaced in UM version 6.11 and beyond by Transport Services Provider (XSP).

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 (receiver)** set to 1 use a thread from the thread pool instead of the context thread. The option, **receive_thread_pool_size (context)** 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 (receiver)** 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 (receiver)** 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 (context)** - **request_tcp_port_high (context)** 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 or Persistent receivers (UMP/UMQ products).
- Multi-Transport Threads do not support or queuing receivers (UMQ product).
- Multi-Transport Threads are not compatible with UMDS Server or UMCache

2.8 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:

- A received message on a topic that the application has subscribed to.
- A timer expiring. Applications can schedule timers to expire in a desired number of milliseconds (although the OS may not deliver them with millisecond precision).
- An application-managed file descriptor event. The application can register its own file descriptors with UM to be monitored for state changes (readable, writable, error, etc.).
- New source notification. UM can inform the application when sources are discovered by Topic Resolution.
- Receiver loss. UM can inform the application when a data gap is detected that could not be recovered through the normal retransmission mechanism.
- End of Stream. UM can inform a receiving application when a data stream (Transport Session) has terminated.

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.

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.

2.9 Rate Controls

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

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

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

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

2.9.1 Transport Rate Control

With data transport, if an outgoing datagram would exceed the number of bits allowed during the current time interval, that datagram is queued and the transport type is put into a "blocked" state in the current context. Any subsequent sends within the same time interval will not queue, but instead will either block (for blocking sends), or return **LBM_EWOULDBLOCK** (for non-blocking sends). When the time interval expires, the context thread will refresh the number of allowable bits, send the queued datagram, and unblock the transport type.

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

Configuration parameters of interest are:

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

2.9.2 Topic Resolution Rate Control

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

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

For details of Topic Resolution, see Topic Resolution Description.

2.10 Operational Statistics

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

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

Chapter 3

UM Objects

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.

3.1 Context Object

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

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

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

Warning

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

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

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

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

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

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

Attention

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

3.2 Topic Object

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

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

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

3.3 Source Object

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

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

3.3.1 Source String

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

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

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

- TCP:src_ip:src_port:session_id[topic_idx]
  session_id is optional, per configuration option **transport_tcp_use_session_id (source)**
  example: `TCP:192.168.0.4:45789:f1789bcc[1539853954]`
- LBTRM:src_ip:src_port:session_id:mc_group:dest_port[topic_idx]
  example: `LBTRM:10.29.3.88:14390:e0679abb:231.13.13.13:14400[1539853954]`
- LBT-RU:src_ip:src_port:session_id[topic_idx]
  session_id is optional, per configuration option **transport_lbtru_use_session_id (source)**
  example: `LBT-RU:192.168.3.189:34678[1539853954]`
- LBT-IPC:session_id:transport_id[topic_idx]
  example: `LBT-IPC:6481f8d4:20000[1539853954]`
- LBT-SMX:session_id:transport_id[topic_idx]
  example: `LBT-SMX:6481f8d4:20000[1539853954]`
- BROKER
  example: `BROKER`

Please note that the topic index field (topic_idx) may or may not be present depending on your version of UM and/or the setting for configuration option **source_includes_topic_index (context)**.

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

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.

3.3.2 Source Configuration and Transport Sessions

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

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

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

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

3.3.3 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 for information on how to employ ZOD.

3.4 Receiver Object

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

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

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

3.4.1 Receiver Configuration and Transport Sessions

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

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

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

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

3.4.2 UM Wildcard Receivers

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

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

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

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

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

3.4.3 Transport Services Provider Object

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

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

3.4.4 UM Hot Failover Across Contexts Objects

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

For more information, see Hot Failover Across Multiple Contexts.

3.4.5 Zero Object Delivery

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.

3.5 Event Queue Object

A UM event queue object is a serialization queue structure and execution thread for delivery of other objects' events. For example, a Source Object can generate events that the user's application wants to receive via callback. When the source is created, an event queue can be specified as the delivery agent of those events. Multiple UM contexts, sources, and receivers can specify the same event queue, and these events will be delivered in a FIFO manner (first-in, first-out).

Without event queues, these events are delivered via callback from the originating object's context thread, which places the following restrictions on the application callback function being called:

- The application function is not allowed to make certain API calls (mostly having to do with creating or deleting UM objects).
- The application function must execute very quickly without blocking.
- The application does not have control over when the callback executes. 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:

- Your application can set a queue size threshold with **queue_size_warning (event_queue)** and be warned when the queue contains too many messages.
- Your application can set a delay threshold with **queue_delay_warning (event_queue)** and be warned when events have been in the queue for too long.
- The application callback function has no UM API restrictions.
- Your application can control exactly when UM delivers queued events with **lbm_event_dispatch()**. And you can have control return to your application either when specifically asked to do so (by calling **lbm_event_dispatch_unblock()**), or optionally when there are no events left to deliver.
- Your application can take advantage of parallel processing on multiple processor hardware since UM processes asynchronously on a separate thread from your application's processing of received messages. By using multiple application threads to dispatch an event queue, or by using multiple event queues, each with its own dispatch thread, your application can further increase parallelism.

You create an UM event queue in the C API by calling **lbm_event_queue_create()**. When finished with an event queue, delete it by calling **lbm_event_queue_delete()**. See **Event Queue Options** for configuration options related to event queues.

Warning

> Before deleting an event queue, you must first delete all objects that reference that event queue (sources, receivers, wildcard receivers, contexts).

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

3.6 Message Object

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

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

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

3.6.1 Message Object Deletion

C API

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

Java or .NET API

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

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

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

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

3.6.2 Message Object Retention

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

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

C API

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

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

Java or .NET

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

Chapter 4

Transport Types

4.1 Transport TCP

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

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

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

The TCP Session ID enables multiple receivers for a topic to connect to a source across a UM Router. 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.

Note

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

4.2 Transport LBT-RU

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

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

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

The LBT-RU Session ID enables multiple receivers for a topic to connect to a source across a UM Router. 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.

Note

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

4.3 Transport LBT-RM

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

LBT-RM 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 are not flowing. Once data stops on a session, advertisements are sent with an exponential back-off (to a configurable maximum interval) so that the bandwidth taken up by the session is minimal.

- Sender Rate Control - LBT-RM can control a sender's rate of injection of data into the network by use of a rate limiter. This rate is configurable and will back pressure the sender, not allowing the application to exceed the rate limit it has specified. In addition, LBT-RM senders have control over the rate of retransmissions separately from new data. This allows 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.

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

Note

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

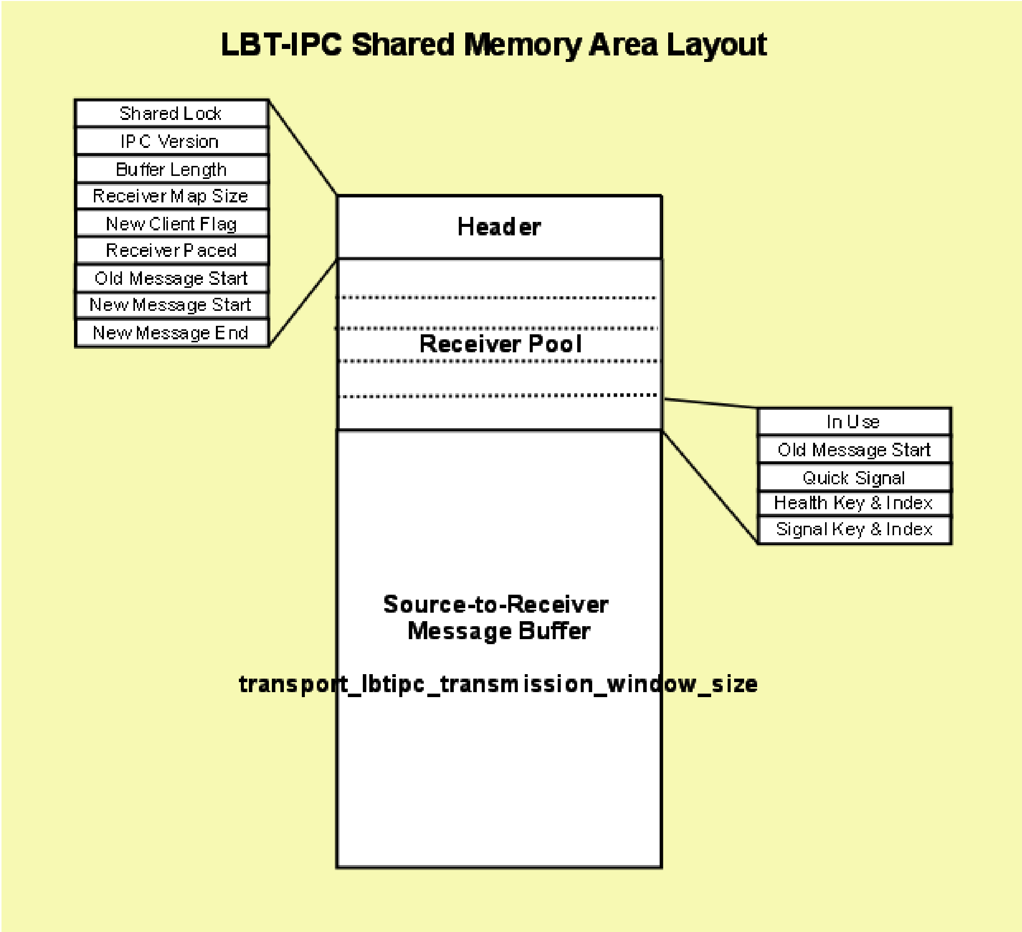
4.4 Transport LBT-IPC

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

4.4.1 LBT-IPC Shared Memory Area

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



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 (source)**.
- 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 (context)** to busy_↩ wait, the receiver sets this semaphore to zero and the source does not notify.

Source-to-Receiver 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 (source)**. 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.

4.4.2 Sources and LBT-IPC

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

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

Using the configuration option, **transport_lbtipc_behavior (source)**, you can choose source-paced or receiver-paced message transport. See **Transport LBT-IPC Operation Options** for more information.

Sending over LBT-IPC

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

4.4.3 Receivers and LBT-IPC

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

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

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

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

Topic Resolution and LBT-IPC

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

Receiver Pacing

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

Receiver Monitoring

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

4.4.4 Similarities with Other UM Transports

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

- If you use a range of LBT-IPC transport IDs, UM assigns multiple topics sent by multiple sources to all the Transport Sessions in a round robin manner just like other UM transports.
- Transport sessions assume the configuration option values of the first source assigned to the Transport Session.
- Sources are subject to message batching.

4.4.5 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 (receiver)** options. However, if you set **ordered_delivery (receiver)** 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.

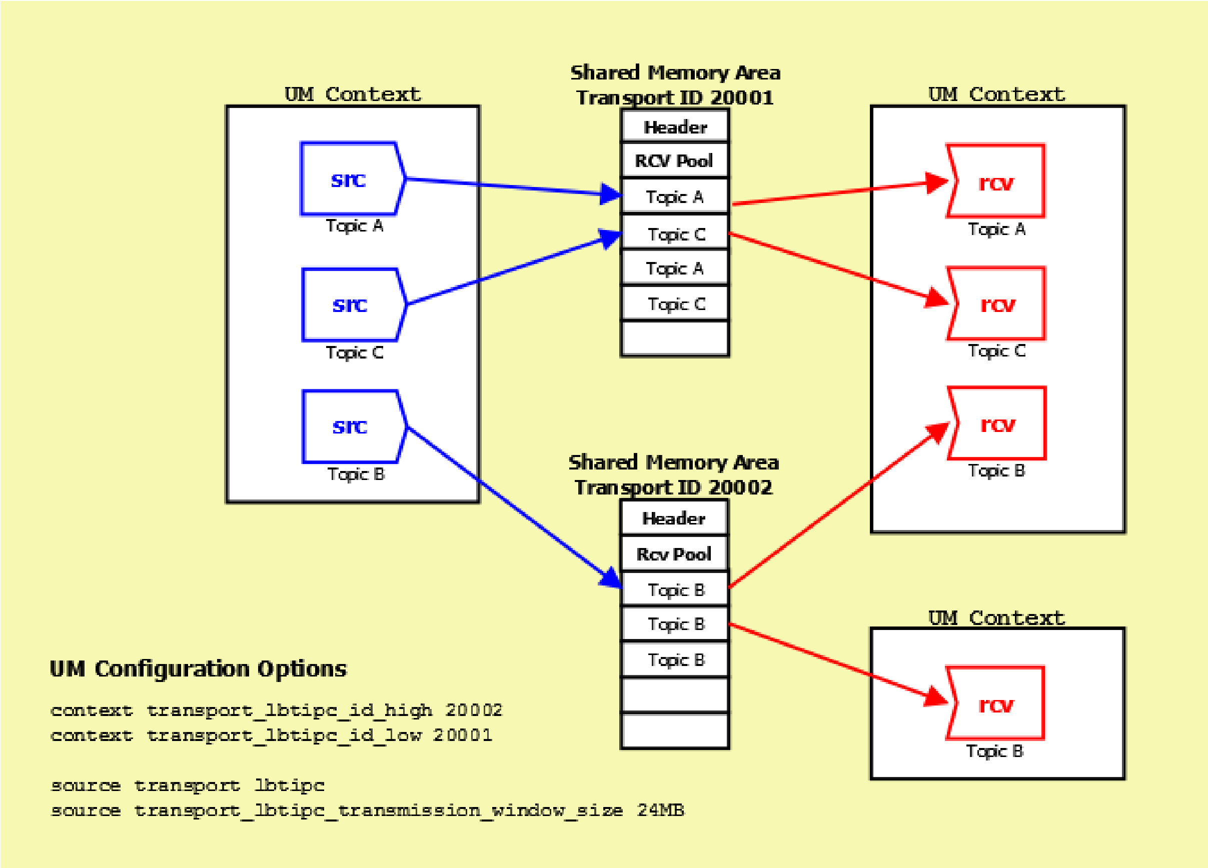
4.4.6 Sending to Both Local and Remote Receivers

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.

4.4.7 LBT-IPC Configuration Example

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



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

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

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

4.4.8 Required privileges

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

4.4.9 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. See also the KB article `Managing LBT-IPC Host Resources`.

4.4.10 LBT-IPC Resource Manager

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

Displaying Resources

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

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

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

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

Reclaiming Unused Resources

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

4.5 Transport LBT-SMX

The LBT-SMX (shared memory acceleration) transport is an Interprocess Communication (IPC) transport you can use for the lowest latency message Streaming. LBT-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.c` and `lbmlatpong.c` show how to use the C LBT-SMX API calls. For Java, see `lbmlatpong.java` and `lbmlatpong.java`. For .NET, see `lbmlatpong.cs` and `lbmlatpong.cs`.

Other example applications can use the LBT-SMX transport with the use of a UM configuration flat 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** for more information.

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

4.5.1 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 (context)** and **transport_lbtsmx_id_low (context)**. You can also specify a shared memory location inside or outside of this range with a source configuration option, **transport_lbtsmx_id (source)**, 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. Example names are **LBTSMX_42792ac_20000** or **LBTSMX_↩ 66e7c8f6_20001**. Receivers access a shared memory area with this object name to receive (read) topic messages.

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.

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.

Note

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.

4.5.2 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 (source)** option minus 16 bytes for headers. In a system deployment that includes the DRO, this value should be the same as the datagram max sizes of other transport types. See **Protocol Conversion**.

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

2. The 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.

4.5.3 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.

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()**.

Note

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 (source)** option minus 16 bytes for headers. Any send API calls with a length parameter greater than this configured value fail. In a system deployment that includes the DRO, this value should be the same as the datagram max sizes of other transport types. See **Protocol Conversion**.

4.5.4 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.

Warning

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

Topic Resolution and LBT-SMX

Topic resolution operates identically with LBT-SMX as other UM transports albeit with the advertisement type, L←BMSMX. 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.

4.5.5 Similarities Between LBT-SMX and Other UM Transports

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

- If you use a range of LBT-SMX transport IDs, UM assigns multiple topics sent by multiple sources to all the Transport Sessions in a round robin manner just like other UM transports.
- Transport sessions assume the configuration option values of the first source assigned to the Transport Session.

- Source applications and receiver applications based on any of the three available APIs can interoperate with each other. For example, sources created by a C sending application can send to receivers created by a Java receiving application.

4.5.6 Differences Between LBT-SMX and Other UM Transports

- Unlike LBT-RM which uses a transmission window to specify a buffer size to retain messages for retransmission, LBT-SMX uses the transmission window option to establish the size of the shared memory. LBT-SMX uses transmission window sizes that are powers of 2. You can set **transport_lbtsmx_transmission_window_size (source)** to any value, but UM rounds the option value up to the nearest power of 2.
- The largest transmission window size for Java applications is 1 GB.
- 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.

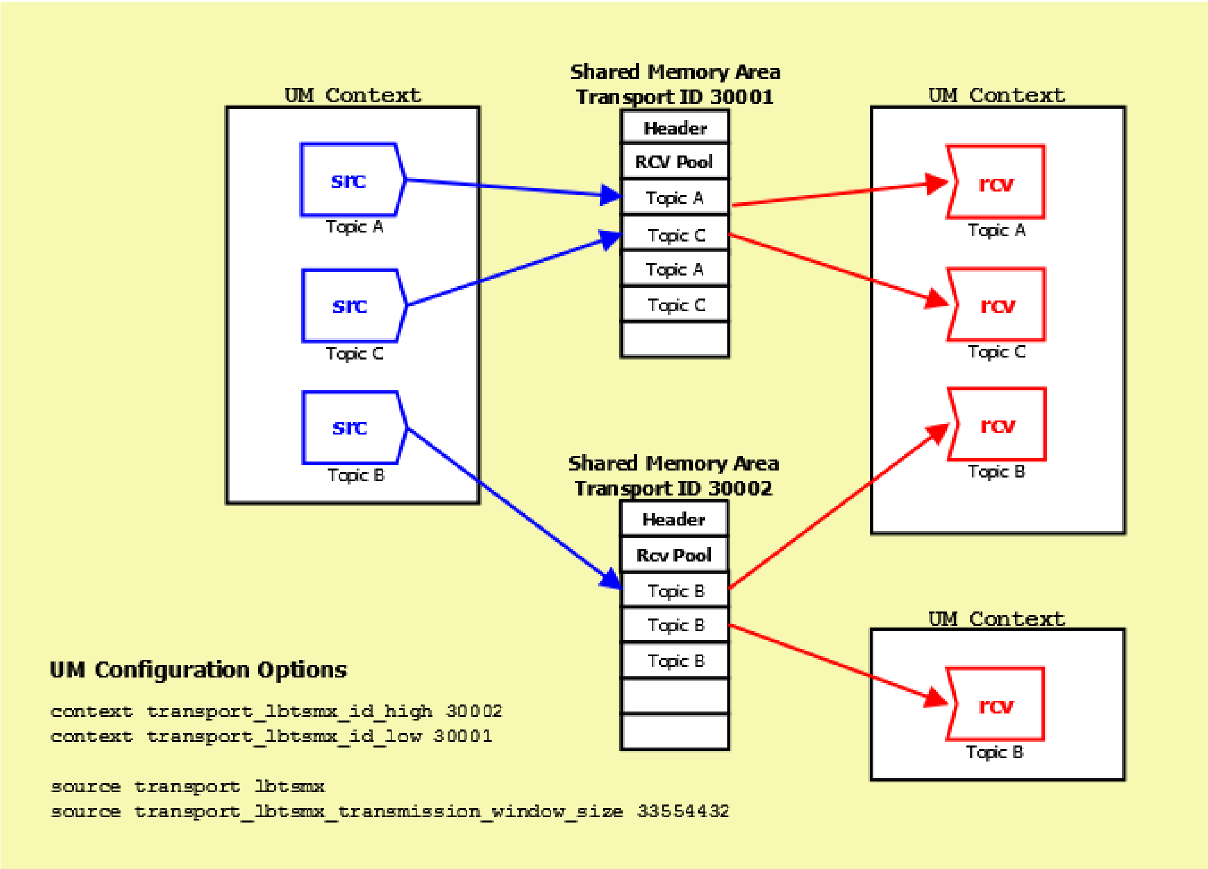
You cannot use the following UM features with LBT-SMX:

- 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"
- Persistent sources
- Queued sources, both brokered and ULB

You also cannot use LBT-SMX to send egress traffic from a UM daemon, such as the Persistent Store, UM Router, UM Cache, or UMDS.

4.5.7 LBT-SMX Configuration Example

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



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

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

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

4.5.8 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();
    }
  } / * main */

  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;/

    * Delay a second to let topic resolution complete. */
    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();
    }/

    * Linger for a short while to allow retransmissions, etc. */
    try { Thread.sleep(1000); } catch (Exception ex) { }

    src.close();
    ctx.close();
  } / * sendMessages */
} / * SimpleSrc */
```

The source sends one million messages using the native LBT-SMX Java APIs. The sendMessages() method obtains a reference to the source's message buffer, which does not change for the life of the source. The method acquire←MessageBufferPosition(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;
  } / * on Receive */

  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();
    }
  } / * main */
}
```

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 Example

```
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 Example

```
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 LBMEWould↩Block 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.

4.5.9 .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.

4.5.10 LBT-SMX Resource Manager

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

Displaying Resources

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

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

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

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

Reclaiming Unused Resources

Warning

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

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

4.6 Transport Broker

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

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

Chapter 5

Topic Resolution Description

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

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

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

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

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

Note

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

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

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

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

TCP-based TR (with "SRS" service) is a more recent addition to UM. It is available as of UM version 6.12, in which it provides a subset of the total TR functionality. Specifically, it supports receivers discovering sources. However, TCP-based TR does not yet support DRO route maintenance and distribution, resolution of Persistent Store names, or redundancy. (These functions will be supported by TCP-based TR in future UM releases.)

For UM version 6.12, TCP-based TR is typically paired with one of the two UDP-based TR protocols. This is done to supply missing TR functionality, and to support interoperability with pre-6.12 versions of UM. The two protocols run in parallel, with the UDP-based TR protocol supplying the missing functionality and providing redundancy to the more-reliable TCP-based TR.

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

5.1 TR Protocol Comparison

5.1.1 Multicast UDP TR

Multicast UDP-based Topic Resolution is the default protocol.

Advantages:

- Very fast source discovery for small deployments.
- Simplicity – no independent service required.
- Highly fault tolerant. No independent services are needed for TR delivery. The internal network infrastructure provides redundancy.

Disadvantages:

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

5.1.2 Unicast UDP TR

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

Advantages:

- Does not use multicast.
- Supports redundant "lbmrd" services, which provides fault tolerance and load balancing.

Disadvantages:

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

5.1.3 TCP TR

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

Advantages:

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

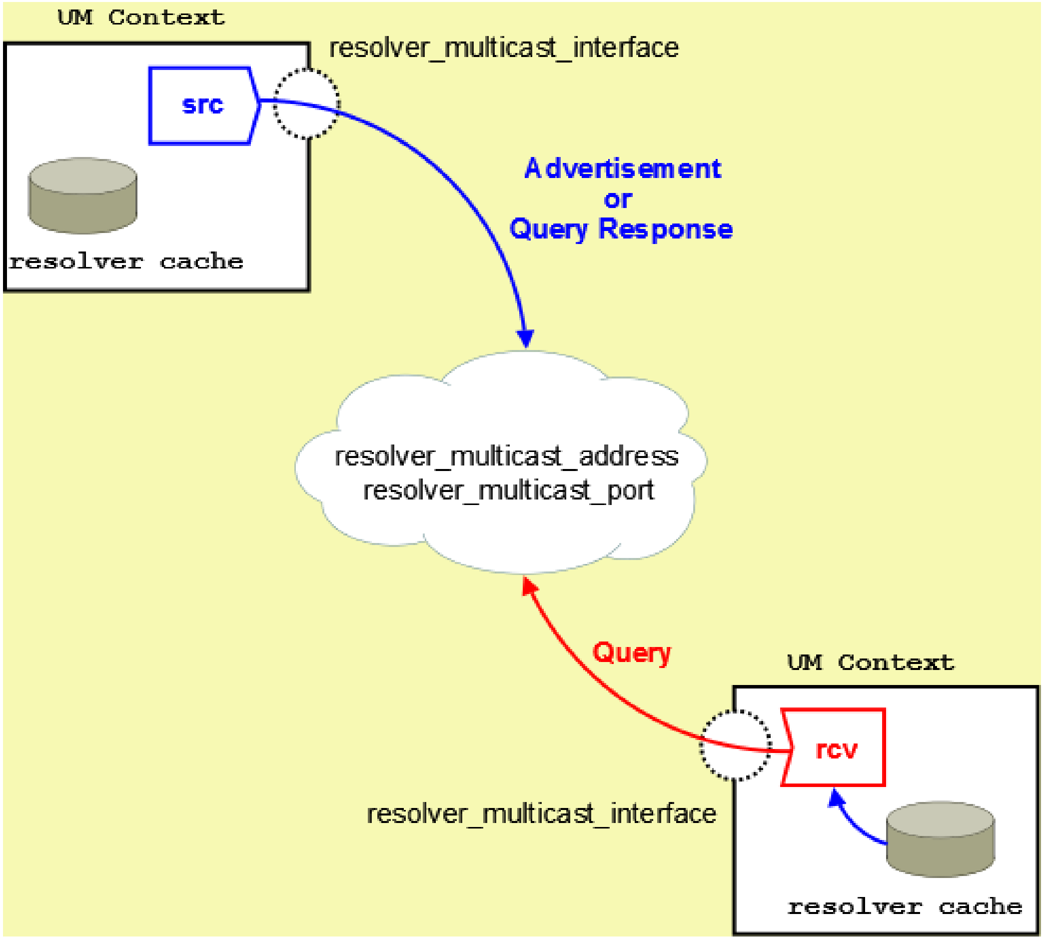
Disadvantages:

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

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

5.2 UDP-Based Topic Resolution Details

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



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

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

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

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

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

UDP-based Topic Resolution occurs in the following phases:

- Initial Phase - Period that allows you to resolve a topic aggressively. This phase can be configured to run differently from the defaults or completely disabled.
- Sustaining Phase - Period that allows new receivers to resolve a topic after the Initial Phase. Can also be the primary period of topic resolution if you disable the Initial Phase. This phase can also be configured to run differently from the defaults or completely disabled.

- Quiescent Phase - The quiet phase where Topic Resolution datagrams are no longer sent in an unsolicited way. This reduces the CPU and network resources consumed by TR, and also reduces latency outliers. However, in large deployments, especially those that include wide-area networks, the Quiescent Phase is sometimes disabled, by configuring the Sustaining Phase to continue forever. This is done to avoid deafness issues.

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

5.2.1 Sources Advertise

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

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

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

- For a TCP transport, the source address, TCP port and Session ID.
- For an LBT-RM transport, the source address, the multicast group address, the UDP destination port, LBT-RM Session ID, and the unicast UDP port to which NAKs are sent.
- For 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-SMX transport, the Host ID, LBT-SMX Session ID and Transport ID.

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

5.2.2 Receivers Query

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

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

A TQR consists primarily of the topic string.

5.2.3 Wildcard Receiver Topic Resolution

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

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

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

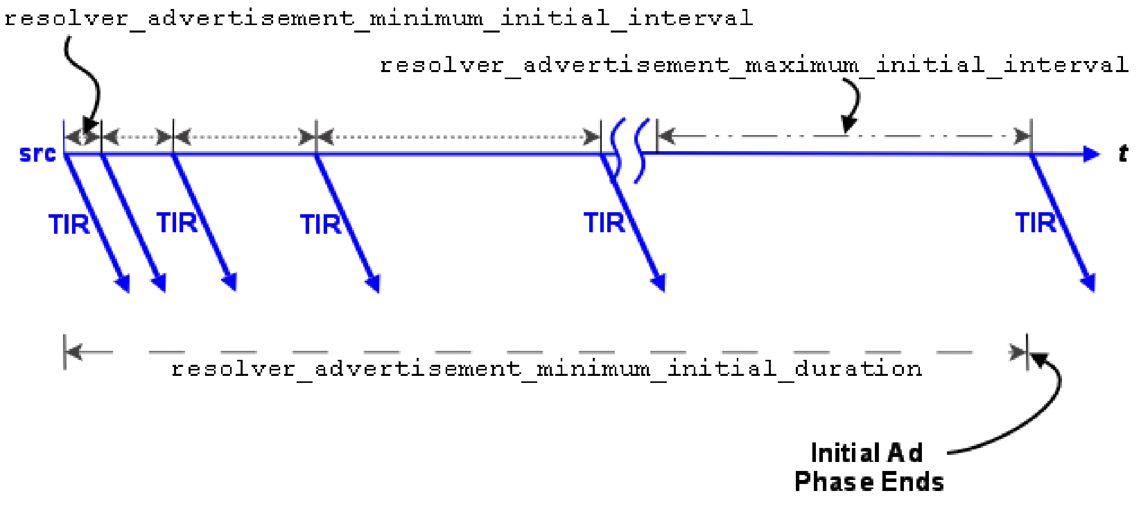
See **Wildcard Receiver Options** for more information.

5.2.4 Initial Phase

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

Advertising in the Initial Phase

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

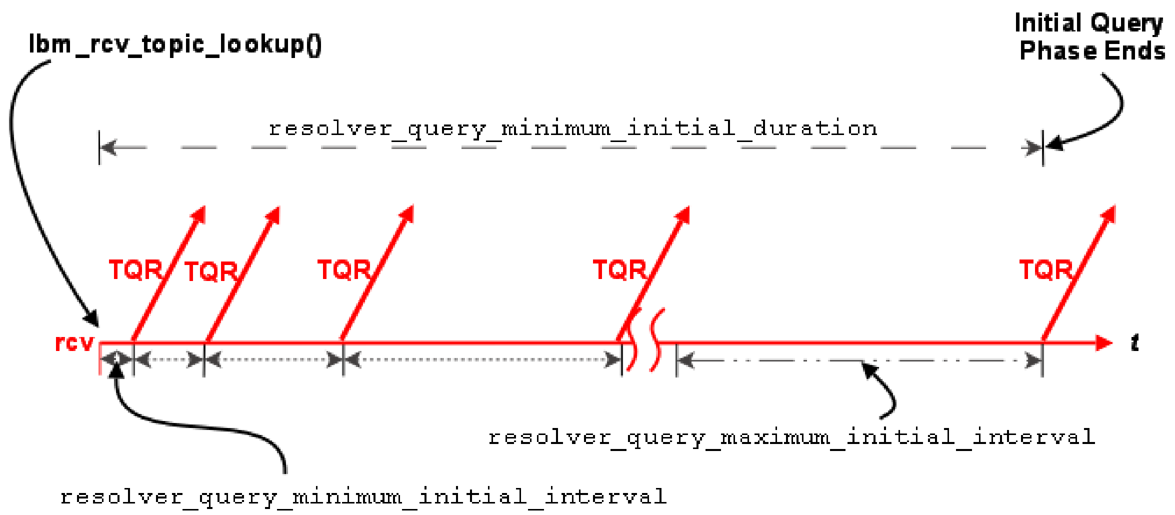


The initial phase for a topic can take longer than the **resolver_advertisement_minimum_initial_duration (source)** if many topics are in resolution at the same time. The configuration options, **resolver_initial_advertisements_per_second (context)** and **resolver_initial_advertisement_bps (context)**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

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

Querying in the Initial Phase

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



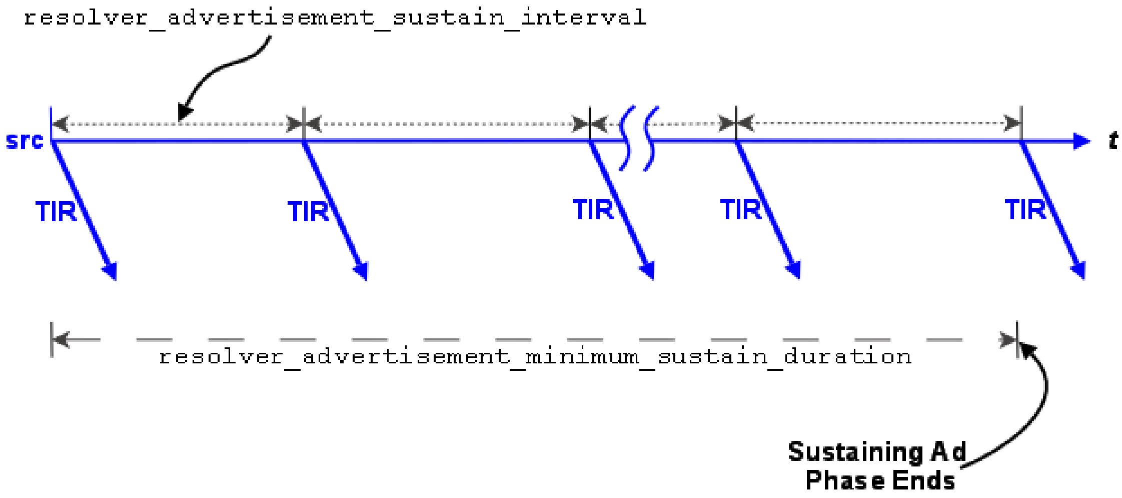
The initial query phase completes when it reaches the **resolver_query_minimum_initial_duration (receiver)**. The initial query phase also has UM context-wide rate limit controls (**resolver_initial_queries_per_second (context)** and **resolver_initial_query_bps (context)**) that can result in the extension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

5.2.5 Sustaining Phase

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

Advertising in the Sustaining Phase

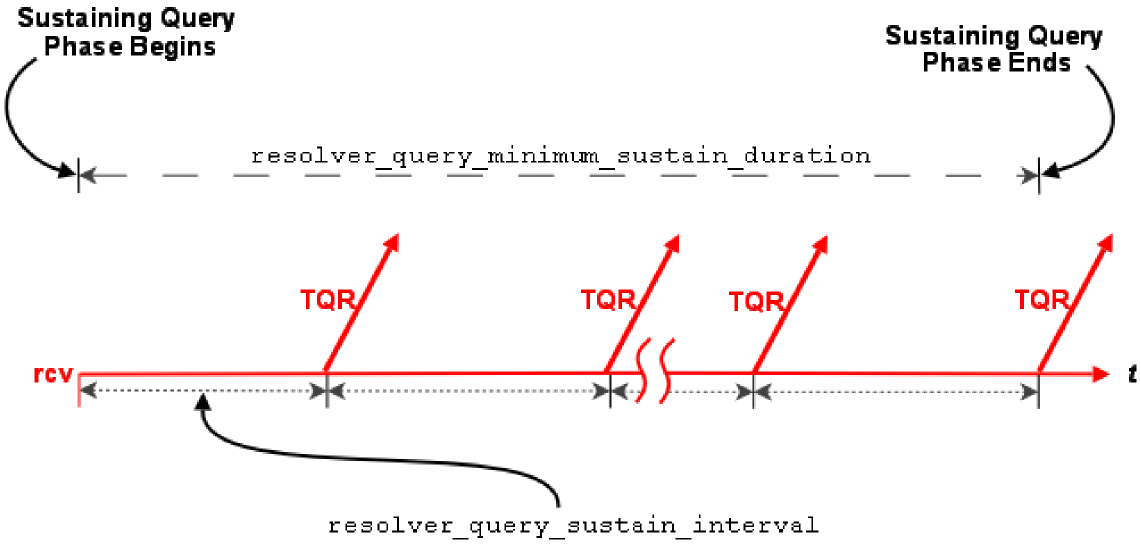
For the sustaining phase defaults, a source sends an advertisement every second (**resolver_advertisement_sustain_interval (source)**) for 1 minute (**resolver_advertisement_minimum_sustain_duration (source)**). When this duration expires, the sustaining phase of advertisement for a topic ends. If a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



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

Querying in the Sustaining Phase

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



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

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

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

5.2.6 Quiescent Phase

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

This phase ends if either of the following occurs.

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

5.2.7 Store (context) Name Resolution

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

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

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

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

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

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

5.2.8 UDP Topic Resolution Configuration Options

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

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

Assigning Different Configuration Options to Individual Topics

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

5.2.9 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 instances of the UM unicast topic resolution daemon (lbmrd), which perform the same topic resolution activities as multicast topic resolution. You configure your applications to use the lbmrd daemons with **resolver_unicast_daemon (context)**.

See Manpage for lbmrd for details on running the lbmrd daemon.

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

- Topic Information Records (TIR) - also known as topic advertisements
- Topic Query Records (TQR)
- keepalive messages, which are only used in unicast topic resolution

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

LBMRD with the UM Router Best Practice

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

Unicast Topic Resolution Resilience

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

5.2.10 Network Address Translation (NAT)

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

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

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

Warning

> Using an lbmrd NAT configuration severely limits the UM features that can be used across the NAT. Normal source-to-receiver traffic is supported, but the following more-advanced UM features are not supported:
>
> - Persistence
> - Queuing
> - Request/Response
> - Late Join
> - Off-Transport Recovery (OTR)
> - Sending to Sources
>
> Late Join, sending to sources, and OTR can be made to work if applications are configured to use the default value (0.0.0.0) for **request_tcp_interface (context)**. This means that you cannot use **default_interface (context)**. Be aware that the UM Router requires a valid interface be specified for **request_tcp_interface (context)**. Thus, **lbmrd NAT support for Late Join, Request/Response, and OTR is not compatible with UM topologies that contain the UM Router**.

5.2.11 Example NAT Configuration

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

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

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

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <interface>10.1.1.50</interface>
    <port>12000</port>
  </daemon>
  <domains>
    <domain name="Net-NYC">
      <network>10.1.0.0/16</network>
    </domain>
    <domain name="Net-NJC">
      <network>192.168.1/24</network>
    </domain>
  </domains>
  <transformations>
```

```
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        <match address="10.1.1.50" port="*"/>
        <replace address="192.168.1.1" port="*"/>
      </rule>
    </transform>
  </transformations>
</lbmrd>
```

The lbmrd must be run on 10.1.1.50.

The application on 10.1.1.50 should be configured with:

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

The applications in the 192 network should be configured with:

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

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

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

5.3 UDP-Based Topic Resolution Strategies

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

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

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

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

5.3.1 Default TR

The main characteristics of UM's default TR settings are:

- Multicast UDP.
- Three phases enabled (Initial, Sustaining, Quiescent). Unsolicited TIRs and TQRs nominally last for 65 seconds, although that number can grow as the number of sources or receivers in a context increases.

The default settings can be fine for reasonably small, static deployments, typically not including Wide Area Networks. (A "static" deployment is one where sources, and receivers are, for the most part, created during system startup, and deleted during system shutdown. Contrast with a "dynamic" system where applications come and go during normal operation, with sources and receivers being created and deleted at unpredictable times.)

Advantages:

- Simplicity.
- In a network where sources and receivers are relatively static, the consumption of resources by TR stops reasonably quickly.

Disadvantages:

- As the numbers of contexts, sources, and receivers grow, the traffic load during the initial phase can be very intense, leading to packet loss and potential deafness issues. In these cases, the initial phase can be configured to be less aggressive, or disabled altogether.
- If a network outage lasts longer than 65 seconds, it is possible for new sources and receivers to be deaf to each other, due to entering their quiescent phases. In these cases, the sustaining phase can be configured for longer durations.

5.3.2 Query-Centric TR

The main characteristics of Query-centric TR are:

- Unsolicited TIRs are severely limited or disabled. See **Disabling Aspects of Topic Resolution**.
- TQRs are extended, often to infinity.

Query-centric TR can be useful for large-scale, dynamic systems, especially those that may have many sources for which there are no receivers during normal operation. For example, in some market data distribution architectures, many tens of thousands of sources are created, but a fairly small percentage of them have receivers at any given time. In that case, it is unnecessary to advertise sources on topics that have no receivers.

Note that this strategy does not prevent advertisements. Each TQR will trigger one or more sources to send a TIR in response.

Advantages:

- For some deployments, can result in significantly reduced TR loading due to removal of TIRs for topics with no receivers.

Disadvantages:

- To avoid deafness issues, the Query sustaining phase is usually extended, often to infinity. This consumes CPU and Network bandwidth, and can introduce latency outliers.

- For topics that have receivers, both TQR and TIR traffic are present. (In contrast, a Advertise-Centric TR strategy removes the TQRs, but at the expense of advertising *all* sources, even those that have no receivers.)

5.3.3 Known Query Threshold TR

In a special case of Query-centric TR, certain classes of topics have a specific number of sources. For example, in point-to-point use cases, a particular topic has exactly one source. As another example, some market data distribution architectures have two sources for each topic, a primary and a warm standby.

For those topics where it is known how many sources there should be, the configuration option **resolution_number_of_sources_query_threshold (receiver)** can be combined with Query-centric TR to great benefit.

For example, consider a market data system with a primary and warm standby source for each topic. Unsolicited advertisements are disabled (see **Disabling Aspects of Topic Resolution**), and **resolution_number_of_sources_query_threshold (receiver)** is set to 2. The receiver will query until it has discovered two sources, at which point it will stop sending queries. If a source fails, the receiver resumes sending queries until it again has two sources.

The advantage here is that it is no longer necessary to extend the Sustaining phase forever to avoid deafness.

NOTE: wildcard receivers do not fit well with this model of TR. Wildcard receivers have their own query mechanism; see Wildcard Receiver Topic Resolution. In particular, there is no wildcard equivalent to the number of sources query threshold. In a query-centric model, wildcard queries must be extended to avoid potential deafness issues. However, in most deployments, the number of wildcard receiver objects is small compared to the number of regular single-topic receivers, so using the Known Query Threshold TR model can still be beneficial.

5.3.4 Advertise-Centric TR

The main characteristics of Advertise-centric TR are:

- Unsolicited TQRs are severely limited or disabled. See **Disabling Aspects of Topic Resolution**.
- TIRs are extended, often to infinity.

Advertise-centric TR can be useful for large-scale, dynamic systems, especially those that may have very few sources for which there are no receivers. For example, most order management and routing systems use messaging in a point-to-point fashion, and every source should have a receiver. In that case, it is unnecessary to extend queries.

Advantages:

- For some deployments, can result in moderate reduced TR loading due to reduction of TQRs.

Disadvantages:

- To avoid deafness issues, the Advertising sustaining phase is usually extended, often to infinity. This consumes CPU and Network bandwidth, and can introduce latency outliers.
- For topics that have no receivers, TIR traffic is present. (In contrast, a Query-Centric TR strategy removes the TIRs for topics that have no receivers, but at the expense of introducing both TQRs and TIRs.)
- In a deployment that includes the UM Router, some number of TQRs are necessary to inform the Router that the context is interested in the topic. To avoid deafness issues, it is recommended to extend the Querying Sustaining Phase, although at a reduced rate.

5.4 TCP-Based Topic Resolution Details

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

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

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

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

The basic approach used by TCP-based TR is as follows: Each context in a TRD is configured with the address of an SRS service. When the context is created, it connects to the SRS service. When the connection is successful, the context and SRS exchange TR information. They normally do this without involving the other contexts in the TRD.

Then, as an application creates or deletes sources, its context informs the SRS of the change, which in turn informs the other contexts in the TRD.

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

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

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

UM version 6.12 provides an initial implementation of TCP-based TR, and is able to significantly reduce CPU and network loading, reduce latency outliers, and avoid TR-induced packet loss. However, future versions of UM will enhance TCP-based TR significantly, leading to even greater increases in scaling and reductions in load.

A note about the term "stateful" in relation to the SRS. Even though Unicast UDP TR uses a service called "lbmrd", that service does not maintain the topic information. Instead, the "lbmrd" service merely forwards TR datagrams received, essentially simulating Multicast. For a newly-started receiving application to discover an existing source, that source must send a new TIR to the "lbmrd", which in turn forwards it to the new receiver.

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

5.4.1 TCP-Based TR and Fault Tolerance

A limitation of UM version 6.12 TCP-based TR is its lack of redundancy in the SRS service. Many users will want a backup to the SRS; in UM 6.12 that backup is UDP-based TR.

Note that in a single TRD environment with no UM Router, failure of TR only affects resolution of new sources and receivers. Existing data streams will continue uninterrupted. So some users may opt to run their system without UDP-based TR to gain the full benefits of TCP-based TR. These users simply ensure that a failed SRS is restarted in a reasonably timely way.

However, many users desire the elimination of single-points of failure, and will therefore need to run TCP-based TR in parallel with UDP-based TR.

Fortunately, the benefits of TCP-based TR can still be largely gained by reducing the amount of UDP-based traffic. The principal behind this is as follows: any form of redundancy is intended to provide a backup service if the primary service fails. It is highly unlikely that both the primary and backup will fail at the same time. In the case of UD↩P-based TR, the extended sustaining phase is intended to handle various UDP failure scenarios. With TCP-based TR as the primary and UDP-based TR as the secondary, there is no need to extend the sustaining phase since it is highly unlikely that the SRS will fail at the same time that UDP fails.

5.4.2 TCP-Based TR Version Interoperability

TCP-based TR was first introduced in UM version 6.12. To maintain interoperability between pre-6.12 and 6.12, TCP-based TR must be combined with UDP-based TR.

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

Ideally, all applications within a TRD can be upgraded, but this is often not possible. How can the TR load be reduced in a step-wise fashion while an organization is upgrading applications gradually, over a long period of time?

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

Some helpful strategies might be:

- Identify those topics or classes of topics that have limited application interest. If topic X has sources and receivers in upgraded applications, the UDP-based TR for that topic can be reduced (e.g. sustaining phase greatly reduced).
- Identify those TRDs that have small numbers of applications. When a given TRD's applications have all been upgraded, the UDP-based TR for *all* topics in that TRD can be reduced. If practical, applications can be moved between TRDs to enable some TRDs to be populated by UM version 6.12 and beyond. Also, a TRD can be sub-divided, separating pre-upgraded from post-upgraded.

5.4.3 TCP-Based TR Configuration

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

```
context resolver_service 10.29.3.41:12000
```

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

```
context resolver_service test1.29west.com:12000
```

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

5.4.4 SRS Service

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

See Manpage for SRS for details on running the SRS service.

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

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

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

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

SRS State Lifetime

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

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

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

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

SRS Log File

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

SRS Rolling Logs

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

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

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

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

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

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

Let's say that the log file grows at 1 million bytes per hour (VERY unlikely for an SRS, but let's assume for illustration purposes). At 11:00 pm, the log file will reach 23 million bytes, and will roll. Then, at 12:00 midnight, the log file will roll again, even though it is only 1 million bytes in size.

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

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

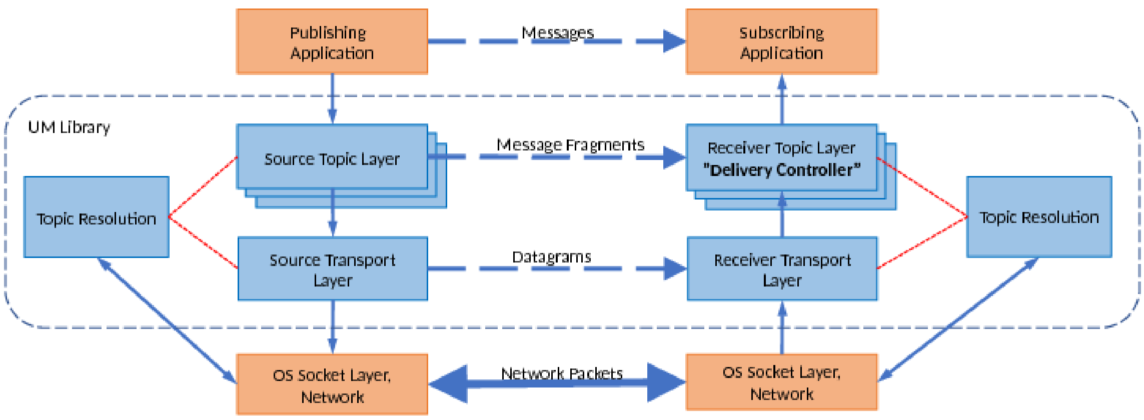
Chapter 6

Architecture

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

6.1 UM Software Stack

Here is a simplified diagram of the software stack:



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

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

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

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

6.1.1 Delivery Controller

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

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

In most communication stacks, the transport layer is responsible for both reliability and ordering - ensuring that messages are delivered in the same order that they were sent. The UM division of functionality is different. It is the Delivery Controller which re-orders the datagrams into the order originally sent. The transport layer delivers datagrams to the Delivery Controller in the order that they arrive. If there is datagram loss, the Delivery Controller sees a gap in the series of topic messages. It buffers the post-gap messages until transport layer arranges retransmission and gives the retransmitted datagrams to the Delivery Controller. The Delivery Controller will then deliver to the application the re-transmitted message, followed by the buffered messages in proper order to the application.

This is an important feature because if a datagram is lost and requires retransmission, that topic will experience additional latency since it takes time for that handshake to happen. However, because the Transport Layer delivers datagrams as they arrive, the Delivery Controller is able to deliver messages for topics that are unaffected by the loss. See Example: Loss Recovery for an illustration of this.

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

6.2 Embedded Mode

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

6.3 Sequential Mode

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

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

- The LBT-IPC transport type, when used, creates its own specialized receive thread. Similar to the context thread, the creation of this thread can be suppressed by setting the option **transport_lbtipc_receiver_↩ operational_mode (context)** to sequential. The application must then call **lbm_context_process_lbtipc↩ _messages()** regularly.
- The LBT-SMX transport type, when used, creates its own specialized receive thread. However, unlike the context thread and the LBT-IPC threads, the creation of the LBT-SMX thread is handled by UM. There is no sequential mode for the LBT-SMX thread.
- The **DBL transport acceleration**, when used, creates its own specialized receive thread. However, unlike the context thread and the LBT-IPC threads, the creation of the DBL thread is handled by UM. There is no sequential mode for the DBL thread.

6.4 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.

With the UMQ product, you cannot use these message batching features with Brokered Queuing.

6.4.1 Implicit Batching

UM automatically batches smaller messages into Transport Session datagrams. The implicit batching configuration options, **implicit_batching_interval (source)** (default = 200 milliseconds) and **implicit_batching_minimum_↩ length (source)** (default = 2048 bytes) govern UM implicit message batching. Although these are source options, they actually apply to the Transport Session to which the source was assigned.

See **Implicit Batching Options**.

See also Source Configuration and Transport Sessions.

UM establishes the implicit batching parameters when it creates the Transport Session. Any sources assigned to that Transport Session use the implicit batching limits set for that Transport Session, and the limits apply to any and all sources subsequently assigned to that Transport Session. This means that batched transport datagrams can contain messages on multiple topics.

Implicit Batching Operation

Implicit Batching buffers messages until:

- the buffer size exceeds the configured **implicit_batching_minimum_length (source)**, or

- the oldest message in the buffer has been in the buffer for **implicit_batching_interval (source)** milliseconds, or
- adding another message would cause the buffer to exceed the configured maximum datagram size for the underlying transport type (transport_∗_datagram_max_size).

When at least one condition is met, UM flushes the buffer, pushing the messages onto the network.

Note that the two size-related parameters operate somewhat differently. When the application sends a message, the **implicit_batching_minimum_length (source)** option will trigger a flush *after* the message is sent. I.e. a sent datagram will typically be larger than the value specified by **implicit_batching_minimum_length (source)** (hence the use of the word "minimum"). In contrast, the transport_∗_datagram_max_size option will trigger a flush *before* the message is sent. I.e. a sent datagram will never be larger than the transport_∗_datagram_max_size option. If both size conditions apply, the datagram max size takes priority. (See **transport_tcp_datagram_max_size (context)**, **transport_lbtrm_datagram_max_size (context)**, **transport_lbtru_datagram_max_size (context)**, **transport_lbtipc_datagram_max_size (context)**, **transport_lbtsmx_datagram_max_size (source)**.)

It may appear this design introduces significant latencies for low-rate topics. However, remember that Implicit Batching operates on a Transport Session basis. Typically many low-rate topics map to the same Transport Session, providing a high aggregate rate. The **implicit_batching_interval (source)** option is a last resort to prevent messages from becoming stuck in the Implicit Batching buffer. If your UM deployment frequently uses the **implicit_batching_interval (source)** to push out the data (i.e. if the entire Transport Session has periods of inactivity longer than the value of **implicit_batching_interval (source)** (defaults to 200 ms), then either the implicit batching options need to be fine-tuned (reducing one or both), or you should consider an alternate form of batching. See Intelligent Batching.

The minimum value for the **implicit_batching_interval (source)** is 3 milliseconds. The actual minimum amount of time that data stays in the buffer depends on your Operating System and its scheduling clock interval. For example, on a Solaris 8 machine, the actual time is can be as much as 20 milliseconds. On older Microsoft Windows machines, the time can be as much as 16 milliseconds. On a Linux 2.6 kernel, the actual time is 3 milliseconds (+/-1).

Implicit Batching Example

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

```
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 trade off between latency and efficiency, where efficiency affects the highest throughput attainable. In general, a large minimum length setting increases efficiency and allows a higher peak message rate, but at low message rates a large minimum length can increase latency. A small minimum length can lower latency at low message rates, but does not allow the message rate to reach the same peak levels due to inefficiency. An intelligent use of implicit batching and application-level flushing can be used to implement an adaptive form of batching known as Intelligent Batching which can provide low latency and high throughput with a single setting.

6.4.2 Intelligent Batching

Intelligent Batching uses Implicit Batching along with your application's knowledge of the messages it must send. It is a form of dynamic adaptive batching that automatically adjusts for different message rates. Intelligent Batching can provide significant savings of CPU resources without adding any noticeable latency.

For example, your application might receive input events in a batch, and therefore know that it must produce a corresponding batch of output messages. Or the message producer works off of an input queue, and it can detect messages in the queue. In any case, if the application knows that it has more messages to send without going to sleep, it simply does normal sends to UM, letting Implicit Batching send only when the buffer meets the **implicit_batching_minimum_length (source)** threshold.

However, when the application detects that it has no more messages to send after it sends the current message, it sets the FLUSH flag (LBM_MSG_FLUSH) when sending the message which instructs UM to flush the implicit batching buffer immediately by sending all messages to the transport layer. Refer to **lbm_src_send()** in the UM API documentation (UM C API, UM Java API, or UM .NET API) for all the available send flags.

When using Intelligent Batching, it is usually advisable to increase the **implicit_batching_minimum_length (source)** option to 10 times the size of the average message, to a maximum value of 8196. This tends to strike a good balance between batching length and flushing frequency, giving you low latencies across a wide variation of message rates.

6.4.3 Application Batching

In all of the above situations, your application sends individual messages to UM and lets UM decide when to push the data onto the wire (often with application help). With application batching, your application buffers messages itself and sends a group of messages to UM with a single send. Thus, UM treats the send as a single message. On the receiving side, your application needs to know how to dissect the UM message into individual application messages.

This approach is most useful for Java or .NET applications where there is a higher per-message cost in delivering an UM message to the application. It can also be helpful when using an event queue to deliver received messages. This imposes a thread switch cost for each UM message. At low message rates, this extra overhead is not noticeable. However, at high message rates, application batching can significantly reduce CPU overhead.

6.4.4 Explicit Batching

UM allows you to group messages for a particular topic with explicit batching. The purpose of grouping messages with explicit batching is to allow the receiving application to detect the first and last messages of a group without needing to examine the message contents.

Note

> Explicit Batching does not guarantee that all the messages of a group will be sent in a single datagram.

Warning

> Explicit Batching does not provide any kind of transactional guarantee. It is possible to receive some messages of a group while others are unrecoverably lost. If the first and/or last messages of a group are unrecoverably lost, then the receiving application will not have an indication of start and/or end of the group.

When your application sends a message (**lbm_src_send()**) it may flag the message as being the start of a batch (LBM_MSG_START_BATCH) or the end of a batch (LBM_MSG_END_BATCH). All messages sent between the start and end are grouped together. The flag used to indicate the end of a batch also signals UM to send the message immediately to the implicit batching buffer. At this point, Implicit Batching completes the batching operation. UM includes the start and end flags in the message so receivers can process the batched messages effectively.

Unlike Intelligent Batching which allows intermediate messages to trigger flushing according to the **implicit_batching_minimum_length (source)** option, explicit batching holds all messages until the batch is completed. This feature is useful if you configure a relatively small **implicit_batching_minimum_length (source)** and your application has a batch of messages to send that exceeds the **implicit_batching_minimum_length (source)**. By releasing all the messages at once, Implicit Batching maximizes the size of the network datagrams.

Explicit Batching Example

The following example demonstrates explicit batching.

```
implicit_batching_minimum_length = 8000
```

1. Your application performs 10 sends of 100 bytes each as a single explicit batch.
2. At the 10th send (which completes the batch), UM delivers the 1000 bytes of messages to the implicit batch buffer.
3. Let's assume that the buffer already has 7899 bytes of data in it from other topics on the same Transport Session
4. UM adds the first 100-byte message to the buffer, bringing it to 7999.
5. UM adds the second 100-byte message, bringing it up to 8099 bytes, which exceeds **implicit_batching_minimum_length (source)** but is below the 8192 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 (source)** is again exceeded, or the **implicit_batching_interval (source)** is met, or a message arrives in the buffer with the flush flag (LBM_MSG_FLUSH) set.

6.4.5 Adaptive Batching

The adaptive batching feature is deprecated and will be removed from the product in a future UM release.

6.5 Message Fragmentation and Reassembly

Message fragmentation is the process by which an arbitrarily large message is split into a series of smaller pieces or *fragments*. Reassembly is the process of putting the pieces back together into a single contiguous message. Ultra Messaging performs fragmentation and reassembly of large user messages. When a user message is small enough, it fits into a single fragment.

Note that there is another layer of fragmentation and reassembly that happens in the TCP/IP network stack, usually by the host operating system. This *IP fragmentation* of datagrams into packets happens when sending datagrams larger than the `MTU` of the network medium, usually 1500 bytes. However, this fragmentation and reassembly happens transparently to and independently of Ultra Messaging. In the UM documentation, "fragmentation" generally refers to the higher-level splitting of messages by the UM library.

Another term that Ultra Messaging borrows from networking is "datagram". In the UM documentation, a *datagram* is a unit of data which is sent to the transport (network socket or shared memory). In the case of network-based transport types, this refers to a buffer which is sent to the network socket in a single system call.

(Be aware that for UDP-based transport types (LBT-RM and LBT-RU), the UM datagrams are in fact sent as UDP datagrams. For non-UDP-based transports, the use of the term "datagram" is retained for consistency.)

The mapping of message fragments to datagrams depends on three factors:

1. User message size,
2. Configured maximum datagram size for the source's transport type, and
3. Use of the Implicit Batching feature.

When configured, the source implicit batching feature combines multiple small user messages into a single datagram no greater than the size of the transport type's configured maximum datagram size. Large user messages are split into N fragments, the first N-1 of which are approximately the size of the transport type's configured maximum datagram size, and the Nth fragment containing the left-over bytes.

Each transport type has its own default maximum datagram size. For example, LBT-RM and LBT-RU have 8K as their default maximum datagram sizes, while TCP and IPC have 64K as their default maximums. These different defaults represent optimal values for the different transport types, and it is usually not necessary to change them. See **transport_tcp_datagram_max_size (context)**, **transport_lbtrm_datagram_max_size (context)**, **transport_lbtru_datagram_max_size (context)**, **transport_lbtipc_datagram_max_size (context)**, **transport_lbtsmx_datagram_max_size (source)**.

Note that the transport's datagram max size option limits the size of the UM *payload*, and does not include overhead specific to the underlying transport type. For example, **transport_lbtrm_datagram_max_size (context)** does not include the UDP, IP, or packet overhead. The actual network frame can be larger than than the configured datagram max size.

Warning

> There is one important circumstance where it is necessary to override one or more defaults to make all datagram max sizes the same, including TCP. In these cases, it is usually best to choose the smallest of the default maximum datagram sizes. See DRO **Protocol Conversion**.

6.5.1 Datagram Max Size and Network MTU

When UM is building the datagram, it reserves an amount of size for the maximum possible UM header. Since most UM messages do not need a large UM header, it is rare for a transport datagram to reach the configured size limit. This can represent a problem for users who configure their systems to avoid IP fragmentation by setting their datagram max size to the MTU of their network: the majority of packets will be significantly smaller than the MTU. Users might be tempted to configure the datagram max size to larger than the MTU to take into account the unused reserved header size, but this is normally not recommended. Some UM message types have different maximum possible UM header, and therefore reserve different amounts of size for the header. A setting that results in most packets being filled close to the network MTU can result in occasional packets which exceed the network MTU, and must be fragmented by the operating system.

For most networks, Informatica recommends setting the datagram max sizes to a minimum of 8K, and allowing the operating system to perform IP fragmentation. It is true that IP fragmentation can decrease the efficiency of network routers and switches, but only if those routers and switches have to perform the fragmentation. With most modern networks, the entire fabric is designed to handle a common MTU, typically of 1500 bytes. Thus, an IP datagram larger than 1500 bytes is fragmented once by the sending host's operating system, and the switches and routers only need to forward the already-fragmented packets. Switches and routers can forward fragmented packets without loss of efficiency.

The only time when it is necessary to limit UM's datagram max size option to an MTU is if a network link in the path has an MTU which is smaller than the host's network interface's MTU. This could be true if an older WAN link is used with an MTU below 1500, or if the host is configured for jumbo frames above 1500, but other links in the network are smaller than that. Because of the variation in UM's reserved size, Informatica recommends setting up networks with a consistent MTU across all links that carry UM traffic.

Kernel Bypass Network Drivers

Users of a kernel bypass network driver frequently want to avoid all IP fragmentation. Some such drivers do not support fragmentation at all, while others do support it but route fragments through the kernel ("slow path"), thus missing the intended performance benefit of the driver.

For applications that need to send messages larger than an MTU, the datagram max size can be reduced so that UM-level fragmentation produces datagrams less than an MTU. However, because UM reserves enough space for the maximum possible header, some users set their datagram max size to a value *above* the MTU. This allows normal LBT-RM and LBT-RU traffic to more-efficiently fill packets (thus reducing packet counts).

However, keep in mind that UM does not publish the internal reserved size, and does not guarantee that the reserved size will stay the same. Users who use this technique determine their optimal datagram max size empirically through testing within the constraints of their use cases.

Finally, for those kernel bypass network drivers that do support "slow-path" IP fragmentation, some users choose to set datagram max sizes that "almost always" avoid IP fragmentation, but will occasionally fragment.

6.6 Ordered Delivery

With the Ordered Delivery feature, a receiver's Delivery Controller can deliver messages to your application in sequence number order or arrival order. This feature can also reassemble fragmented messages or leave reassembly to the application. You can set Ordered Delivery via UM configuration option to one of three modes:

- Sequence Number Order, Fragments Reassembled
- Arrival Order, Fragments Reassembled
- Arrival Order, Fragments Not Reassembled

See **ordered_delivery (receiver)**

Note that these ordering modes only apply to a specific topic from a single publisher. UM does not ensure ordering across different topics, or on a single topic across different publishers. See Message Ordering for more information.

6.6.1 Sequence Number Order, Fragments Reassembled (Default Mode)

In this mode, a receiver's Delivery Controller delivers messages in sequence number order (the same order in which they are sent). This feature also guarantees reassembly of fragmented large messages. To enable sequence number ordered delivery, set the **ordered_delivery (receiver)** configuration option as shown:

```
receiver ordered_delivery 1
```

Please note that ordered delivery can introduce latency when packets are lost (new messages are buffered waiting for retransmission of lost packets).

6.6.2 Arrival Order, Fragments Reassembled

This mode delivers messages immediately upon reception, in the order the datagrams are received, except for fragmented messages, which UM holds and reassembles before delivering to your application. Be aware that messages can be delivered out of order, either because of message loss and retransmission, or because the networking hardware re-orders UDP packets. Your application can then use the sequence_number field of **lbm_msg_t** objects to order or discard messages. But be aware that the sequence number may not always increase by 1; application messages larger than the maximum allowable datagram size will be split into fragments, and each fragment gets its own sequence number. With the "Arrival Order, Fragments Reassembled" mode of delivery, UM will reassemble the fragments into the original large application message and deliver it with a single call to the application receiver callback. But that message's sequence_number will reflect the final fragment.

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

```
receiver ordered_delivery -1
```

6.6.3 Arrival Order, Fragments Not Reassembled

This mode allows messages to be delivered to the application immediately upon reception, in the order the datagrams are received. If a message is lost, UM will retransmit the message. In the meantime, any subsequent messages received are delivered immediately to the application, followed by the dropped packet when its retransmission is received. This mode 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 UM 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.

6.7 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 (source)**. You can also set a time value for the duration that the source sends contiguous TSNIs with configuration option **transport_topic_sequence_number_info_active_threshold (source)**, after which time the source stops issuing TSNIs.

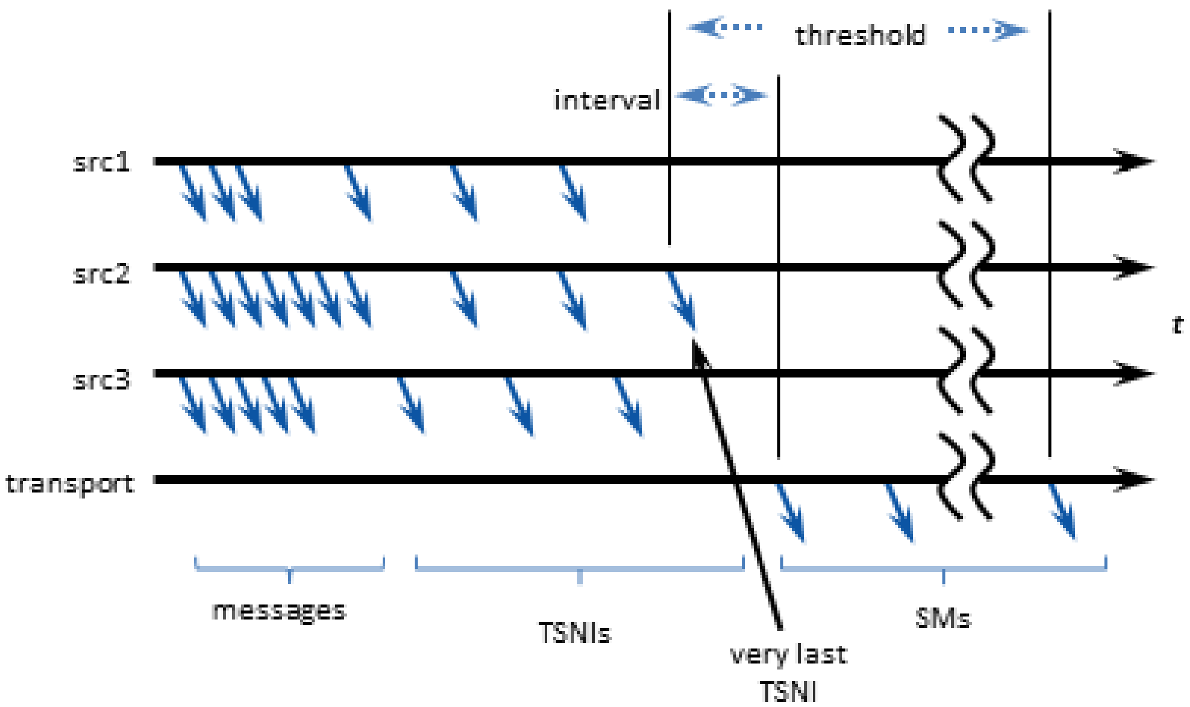
6.8 Receiver Keepalive Using Session Messages

When an LBT-RM, LBT-RU, or LBT-IPC Transport Session enters an inactive period during which it has no messages to send, the UM context sends Session Messages (SMs). The first SM is sent after 200 milliseconds of inactivity (by default). If the period of inactivity continues additional SMs will be sent at increasing intervals, up to a maximum interval of 10 seconds (by default).

SMs serve three functions:

1. "'Keepalive'" - SMs inform receivers that transport sessions are still alive. If a receiver stops getting any kind of traffic for a transport session, after a configurable period of inactivity the receiver will time out the transport session and will assume that it has died.
2. "'Tail loss'" - for UDP-based transport sessions (LBT-RM and LBT-RU), SMs are used to detect packet loss, specifically "tail loss", and trigger recovery.
3. "'Multicast Flows'" - for multicast-based transport sessions (LBT-RM), SMs serve to keep the network hardware multicast flows "hot", so that replication and forwarding of multicast packets is done in hardware at line speed.

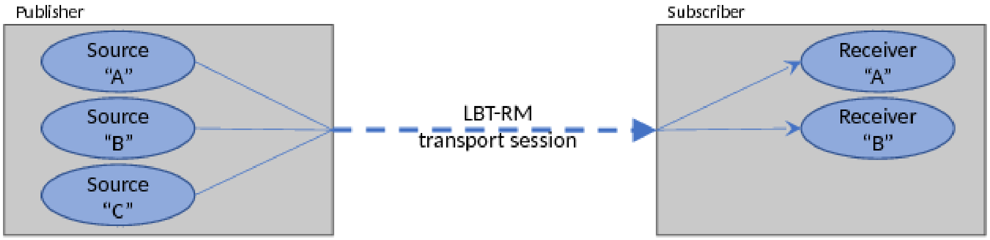
Any other UM message on a transport session will suppress the sending of SMs, including data messages and TSNIs. (Topic Resolution messages are not sent on the transport session, and will not suppress sending SMs.) You can set time values for SM interval and duration with configuration options specific to their transport type.



6.9 Extended Messaging Example

This section illustrates many of the preceding concepts using an extended example of message passing. This example uses LBT-RM, but for the purposes of this example, LBT-RU operates in a similar manner.

The example starts out with two applications, Publisher and Subscriber:



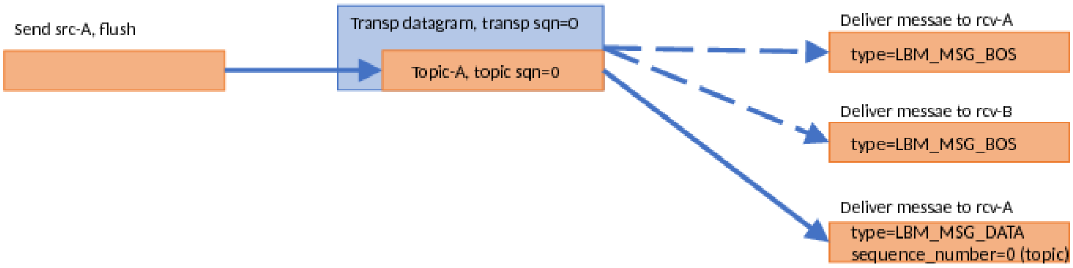
The publisher has created three source objects, for topics "A", "B", and "C" respectively. All three sources are mapped to a single LBT-RM Transport Session by configuring them for the same **multicast group address** and **destination port**.

The Subscriber application creates two receivers, for topics "A" and "B".

The creation of sources and receivers triggers Topic Resolution, and the subscriber joins the Transport Session once the topics are resolved. To be precise, the first receiver to discover a source triggers joining the Transport Session and creating a Delivery Controller; subsequent source discoveries on the same Transport Session don't need to join; they only create Delivery Controllers. However, until such time as one or more publishing sources send their first topic-layer message, the source Transport Session sends no datagrams. The Transport Session is created, but has not yet "started".

6.9.1 Example: First Message

In this example, the first message on the Transport Session is generated by the publishing application sending an application message, in this case for topic "A".



The send function is passed the "flush" flag so that the message is sent immediately. The message is assigned a topic-level sequence number of 0, since it is the application's first message for that topic. The source-side transport layer wraps the application message in a datagram and gives it transport sequence number 0, since it is the first datagram sent on the Transport Session.

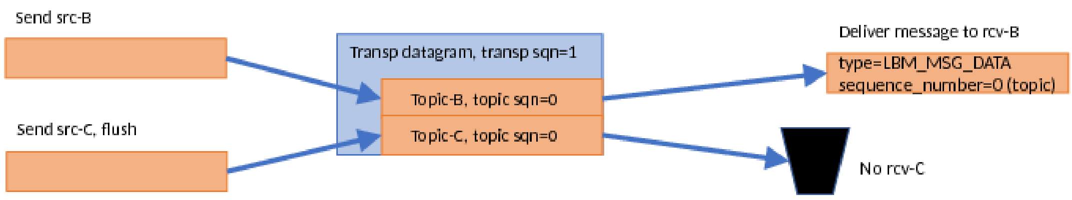
On the receive side, the first datagram (of any kind) on the Transport Session informs the transport layer that the Transport Session is active. The transport layer informs all mapped Delivery Controller instances that the Transport Session has begun. Each Delivery Controller delivers a Beginning Of Session event (BOS) to the application callback for each receiver. The passed-in **lbm_msg_t** structure has event **type** equal to **LBM_MSG_BOS**.

Note that the receiver for topic B gets a BOS even though no messages were received for it; the BOS event informs the receivers that the *Transport Session* is active, not the topic.

Finally, the transport layer passes the received datagram to the topic-A Delivery Controller, which passes the application message to the receiver callback. The passed-in **lbm_msg_t** structure has event **type** equal to **LBM_MSG_DATA**, and a topic-level **sequence_number** of 0. (The transport sequence number is not available to the application.)

6.9.2 Example: Batching

The publishing application now has two more messages to send. To maximize efficiency, it chooses to batch the messages together:



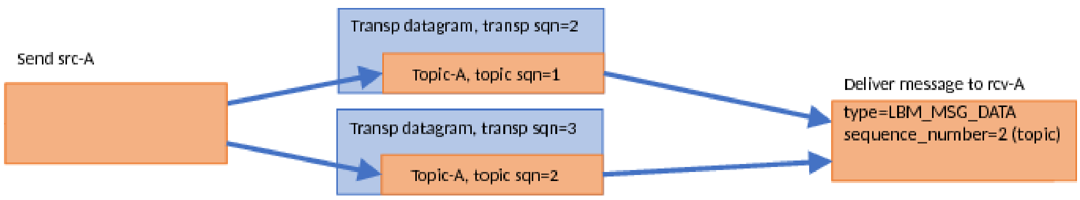
The publishing application sends a message to topic "B", this time *without* the "flush" flag. The source-side topic layer buffers the message. Then the publishing application sends a message to topic "C", *with* the "flush" flag. The source-side transport layer wraps both application messages into a single datagram and gives it transport sequence number 1, since it is the second datagram sent on the Transport Session. But the two topic level sequence numbers are 0, since these are the first messages sent to those topics.

Note that almost no latency is added by batching, so long as the second message is ready to send immediately after the first. This method of low-latency batching is called Intelligent Batching, and can greatly increase the maximum sustainable throughput of UM.

The subscriber gets the datagram and delivers the topic "B" message to the application receiver callback. It's topic-level **sequence_number** is 0 since it was the first message sent to the "B" source. However, the subscriber application has no receiver for topic "C", so the message "C" is simply discarded.

6.9.3 Example: UM Fragmentation

The publishing application now has a topic "A" message to send that is larger than the **maximum allowable datagram**.



The source-side topic layer splits the application message into two fragments and assigns each fragment its own topic-level sequence number (1 for the first, 2 for the second). The topic-layer gives each fragment separately to the transport layer, which wraps each fragment into its own datagram, consuming two transport sequence numbers (2 and 3). Note that the transport layer does not interpret these fragments as parts of a single larger message; from the transport's point of view, this simply two datagrams being sent.

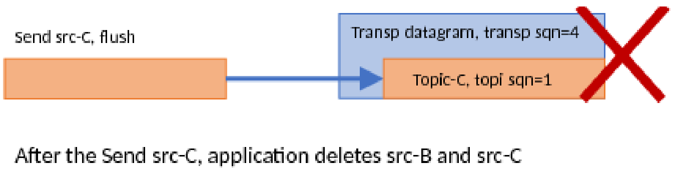
The receive-side transport layer gets the datagrams and hands them to the Topic-A Delivery Controller (receiver-side topic layer). The Delivery Controller reassembles the fragments in the correct order, and delivers the message to the application's receiver callback in a single call. The **sequence_number** visible to the application is the topic-level sequence number of the *last* fragment (2 in this example).

Note that the application receiver callback never sees a topic sequence_number of 1 for topic "A". It saw 0 then 2, with 1 seemingly missing. However, the application can call **lbm_msg_retrieve_fragment_info()** to find out the range of topic sequence numbers consumed by a message.

The behavior described above is for the default **ordered_delivery (receiver)** equal to 1. see Ordered Delivery for alternative behaviors.

6.9.4 Example: Loss Recovery

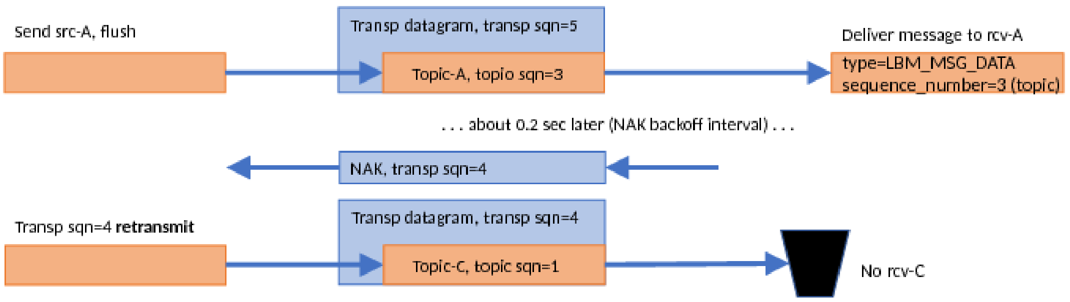
Now the publishing application sends a message to topic C. But the datagram is lost, so the receiver does not see it. Also, right after the send to topic C, the application deletes the sources for topics B and C.



Deleting a source shortly after sending a message to it is contrary to best practice. Applications should pause between the last send to a topic and the deletion of the topic, preferable a delay of between 5 and 10 seconds. This gives receivers an opportunity to attempt recovery if the last message sent was lost. We delete the sources here to illustrate an important point.

Note that although the datagram was lost and two topics were deleted, nothing happens. The receiver does not request a retransmission because the receiver has no idea that the source sent a message. Also, the source-side topic layer does not explicitly inform the receiver that the topics are deleted.

Continuing the example, the publishing application sends another message, this time a message for topic A ("Topic-A, topic sqn=3"):



There are two notable events here:

1. The "A" message is delivered immediately to the topic "A" receiver, even though earlier data was lost and not yet retransmitted. If this were TCP, the kernel would buffer and prevent delivery of subsequent data until the lost data is recovered.

2. The reception of that "A" message with transport sequence number 5 informs the receive-side transport layer that transport datagram #4 was lost. So it initiates a NAK/retransmission cycle. When the lost datagram is retransmitted, the receiver throws it away since it is for an unsubscribed topic.

You might wonder: why NAK and retransmit datagram 4 if the subscriber is just going to throw it away? The subscriber NAKs it because it has no way of knowing which topic it contains; if it were topic B, then it would need that datagram. The publisher retransmits it because it does not know which topics the subscriber is interested in. It has no way of knowing that the subscriber will throw it away.

Regarding message "Topic-A, sqn=3", what if the publisher did not have that message to send? I.e. what if that "Topic-C, sqn=1" message were the last one for a while? This is called "tail loss" since the lost datagram is not immediately followed by a successful datagram. The subscriber has no idea that messages were sent but lost. In this case, the source-side transport layer would have sent a transport-level "session message" after about 200 ms of inactivity on the Transport Session. That session message would inform the receiver-side transport layer that datagram #5 was lost, and would trigger the NAK/retransmission.

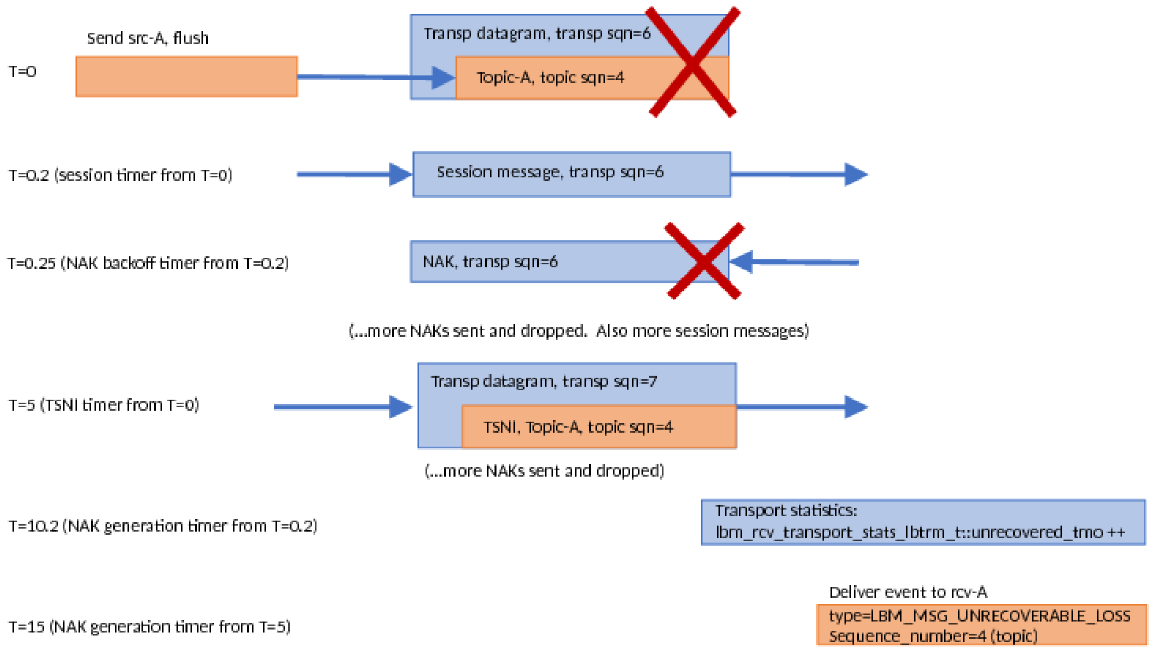
Finally, note that the message for topic-C was retransmitted, even though the topic-C source was deleted. This is because the deletion of a source does not purge the transport layer's retransmission buffer of datagrams from that source. However, higher-level recovery mechanisms, such as late join and OTR, are no longer possible after the source is deleted. Also, if *all* sources on a Transport Session are deleted, the Transport Session itself is deleted, which makes even transport-level retransmission impossible. (Only Persistence allows recovery after the transport session is deleted.)

6.9.5 Example: Unrecoverable Loss

The previous examples assume that events are happening in fairly rapid succession. In this example of unrecoverable loss, significantly longer time periods are involved.

Unrecoverable loss is what happens when UM tries to recover the lost data but it is unable to. There are many possible scenarios which can cause recovery efforts fail, most of which involve a massive overload of one or more components in the data flow.

To simplify this example, let's assume that, starting now, all NAKs are blocked by the network switch. If the publisher never sees the NAKs, it assumes that all datagrams were received successfully and does not retransmit anything.



At T=0, the message "Topic-A, sqn=4" is sent, but not received. Let's assume that the publisher has no more application messages to send for a while. With every application message sent, the source starts two activity timers: a transport-level "session" timer, and a topic-level "TSNI" timer. The session timer is for .2 seconds (see **transport_lbtrm_sm_minimum_interval (source)**), and the TSNI timer is for 5 seconds (see **transport_topic_↩ sequence_number_info_interval (source)**).

At T=0.2, the session timer expires and the source-side transport layer sends a session message. When the receive-side transport layer sees the session message, it learns that transport datagram #6 was lost. So it starts two receive-side transport-level timers: "NAK backoff" and "NAK generation". NAK backoff is shown here as .05 seconds, but is actually randomized between .025 and .075 (see **transport_lbtrm_nak_initial_backoff_interval (receiver)**. The NAK generation is 10 seconds (see **transport_lbtrm_nak_generation_interval (receiver)**).

At T=0.25, the NAK backoff timer expires. Since the transport receiver still has not seen datagram #6, it sends a NAK. However, we are assuming that all NAKs are blocked, so the transport source never sees it. Over the next $\sim$5 seconds, the source will send several more session messages and the receiver will send several more NAKs (not shown).

At T=5, the TSNI timer set by the source at T=0 expires. Since no application messages have been sent since then, the source sends a TSNI message for topic "A". This informs the Delivery Controller that it lost the message "Topic-A, sqn=4". However, the receive-side Delivery Controller (topic layer) does not initiate any recovery activity. It only sets a *topic-level* timer for the same amount of time as the transport's NAK generation timer, 10 seconds. The Delivery Controller assumes that the transport layer will do the actual data recovery.

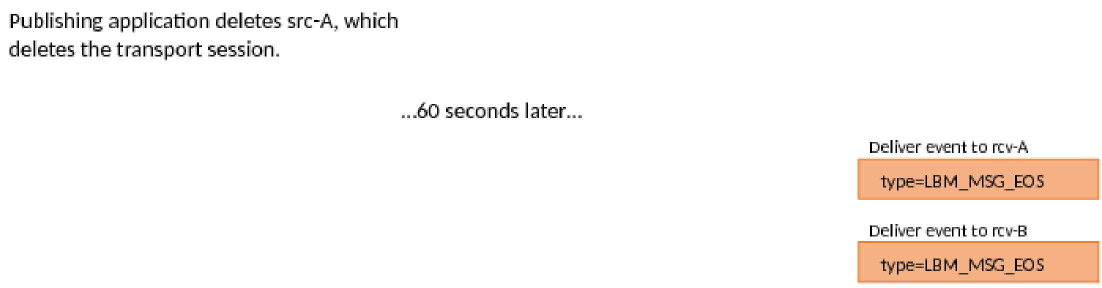
At T=10.2, the receive-side transport layer's NAK generation timer (set at T=0.2) finally expires; the *transport layer* now considers datagram #6 as unrecoverable loss. The transport layer stops sending NAKs for that datagram, and it increments the receive-side transport statistic **lbm_rcv_transport_stats_lbtrm_t_stct::unrecovered_tmo**. Note that it does *not* deliver an unrecoverable loss event to the application.

Over the next $\sim$5 seconds, the Delivery Controller continues to wait for the missing message to be recovered by the transport receiver, but the transport receiver has already given up. You might wonder why the transport layer doesn't inform the Delivery Controller that the lost datagram was unrecoverable loss. The problem is that the transport layer does not know the contents of the lost datagram, and therefore does not know which topic to inform. That is why the Delivery Controller needs to set its own NAK generation timer at the point where it detects topic-level loss (at T=5).

Note that had sources src-B and src-C not been deleted earlier, messages sent to them could have been successfully received and processed during this entire 15-second period. However, any subsequent messages for topic "A" would need to be buffered until T=15. After the unrecoverable loss event is delivered for topic A sequence_number 4, subsequently received and buffered messages for topic "A" are delivered.

6.9.6 Example: Transport Deletion

During the previous 15 seconds, the source-side had sent a number of topic-level TSNI (for topic A) and transport-level session messages. At this point, the publishing application deletes source "A". Since sources "B" and "C" were deleted earlier, "A" was the last source mapped to the Transport Session. So UM deletes the Transport Session.



Note that no indication is sent from the source side to inform receivers of the removal of the sources, nor the Transport Session. So the receive-side transport layer has to time out the Transport Session after 60 seconds of inactivity (see **transport_lbtrm_activity_timeout (receiver)**).

The receive-side transport layer then informs both Delivery Controllers of the End Of Session event, which the Delivery Controllers pass onto the application receiver callback for each topic. The **lbm_msg_t** structure has an event **type** of **LBM_MSG_EOS**. The delivery controllers and the receive-side transport layer instance are then deleted.

However, note that the receiver objects will continue to exist. They are ready in case another publishing application starts up and creates sources for topics A and/or B.

Chapter 7

UM Features

Except where otherwise indicated, the features described in this section are available in the UMS, UMP, and UMQ products.

7.1 Transport Services Provider (XSP)

As of UM version 6.11, a new receive-side object is available to the user: the Transport Services Provider Object.

The earlier feature, Multi-Transport Threads, is deprecated in favor of XSP.

By default, a UM context combines all network data reception into a single *context thread*. This thread is responsible for reception and processing of application messages, topic resolution, and immediate message traffic (UIM and MIM). The context thread is also used for processing timers. This single-threaded model conserves CPU core resources, and can simplify application design. However, it can also introduce significant latency outliers (jitter) if a time-sensitive user message is waiting behind, say, a topic resolution message, or a timer callback.

Using an XSP object, an application can reassign the processing of a subscribed Transport Session to an independent thread. This allows concurrent processing of received messages with topic resolution and timers, and even allows different groups Transport Sessions to be processed concurrently with each other.

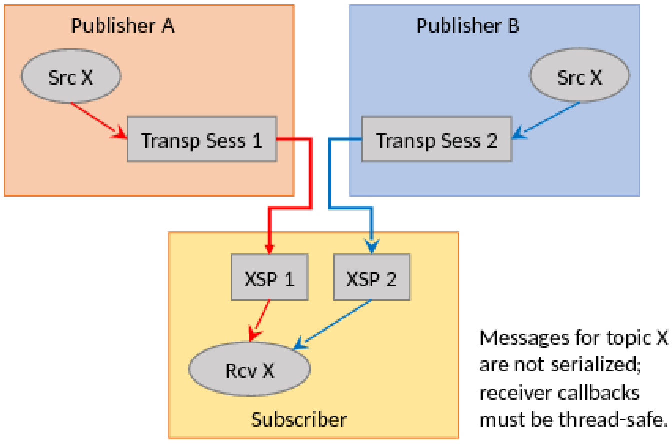
By default, when an XSP object is created, UM creates a new thread associated with the XSP. Alternatively, the XSP can be created with operational mode "sequential", which gives the responsibility of thread creation to the application. Either way, the XSP uses its independent thread to read data from the sockets associated with one or more subscribed Transport Sessions. That thread then delivers received messages to the application via a normal receive application callback function.

Creation of an XSP does not by itself cause any receiver Transport Sessions to be assigned to it. Central to the use of XSPs is an application-supplied mapping callback function which tells UM which XSP to associate with subscribed Transport Sessions as they are discovered and joined. This callback allows the application to examine the newly-joined Transport Session, if desired. Then the callback returns, informing UM which XSP, if any, to assign the receiver Transport Session to.

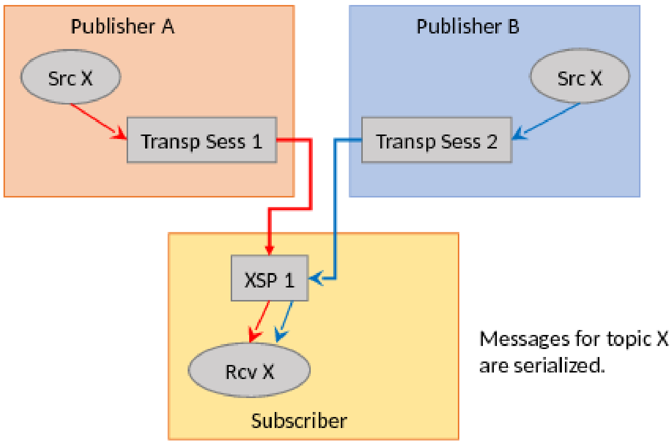
7.1.1 XSP Handles Transport Sessions, Not Topics

Conceptually, an application designer might want to assign the reception and processing of received data to XSPs on a topic basis. This is not always possible. The XSP thread must process received data on a socket basis, and sockets map to *Transport Sessions*. As mentioned in UM Transports, a publishing application maps one or more topic-based sources to a Transport Session.

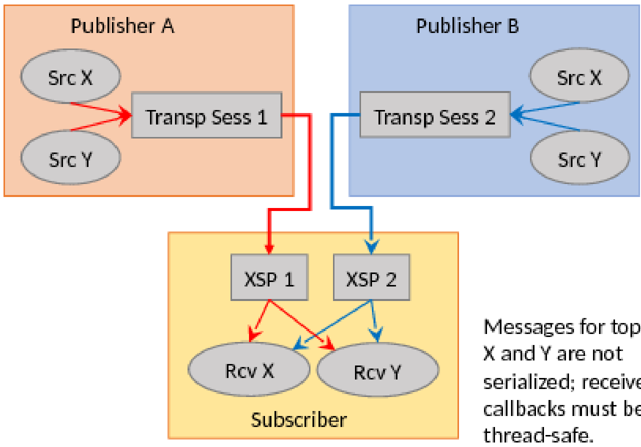
Consider the following example:



Publisher A and B are two separate application instances, both of which create a source for topic "X". A subscriber application might create two XSPs and assign one Transport Session to each. In this case, you have two independent threads delivering messages to the subscriber's receiver callback, which may not be what the developer wanted. If the developer wants topic X to be serialized, a single XSP should be created and mapped to both Transport Sessions:

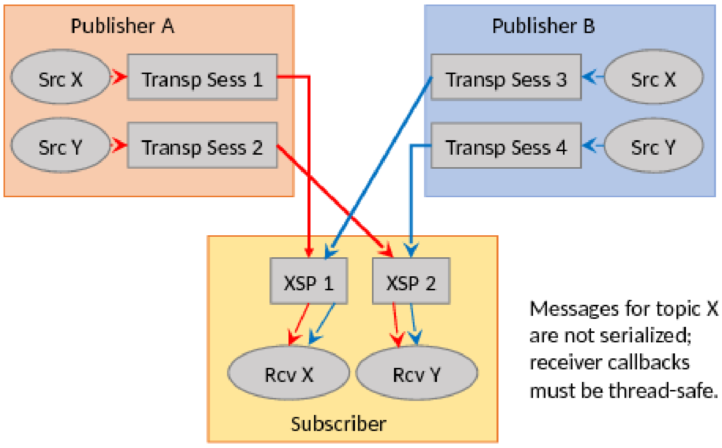


Now let's introduce a second topic. The developer might want to create two XSPs so that each topic will be handled by an independent thread. However, this is not possible, given the way that the topics are mapped to Transport Sessions in the following example:



In this case, XSP 1 is delivering both topics X and Y from Publisher A, and XSP 2 is delivering topics X and Y from Publisher B. Once again, the receiver callback for topic X will be called by two independent threads, which is not desired.

The only way to achieve independent processing of topics is to design the publishers to map their topics to Transport Sessions carefully. For example:



7.1.2 XSP Threading Considerations

When contexts are used single-threaded, the application programmer can assume serialization of event delivery to the application callbacks. This can greatly simplify the design of applications, at the cost of added latency outliers (jitter).

When XSPs are used to provide multi-threaded receivers, care must be taken in application design to account for potential concurrent calls to application callbacks. This is especially true if multiple subscribed Transport Sessions are assigned different XSPs, as demonstrated in XSP Handles Transport Sessions, Not Topics.

Even in the most simple case, where a single XSP is created and used for all subscribed Transport Sessions, there are still events generated by the main context thread which can be called concurrently with XSP callbacks. Reception of MIM or UIM messages, scheduled timers, and some topic resolution-related callbacks all come from the main context thread, and can all be invoked concurrently with XSP callbacks.

Threading Example: Message Timeout

Consider as an example a common timer use case: message timeout. Application A expects to receive messages for topic "X" every 5 seconds. If 10 seconds pass without a message, the application assumes that the publisher for "X" has exited, so it cleans up internal state and deletes the UM receiver object. Each time a message is received, the current timer is cancelled and re-created for 10 seconds.

Without XSPs, this can be easily coded since message reception and timer expiration events are serialized. The timer callback can clean up and delete the receiver, confident that no receiver events might get delivered while this is in progress.

However, if the Transport Session carrying topic "X" is assigned to an independent XSP thread, message reception and timer expiration events are no longer serialized. Publisher of "X" might send it's message on-time, but a temporary network outage could delay its delivery, introducing a race condition between message delivery and timer expiration. Consider the case where the timer expiration is a little ahead of the message callback. The timer callback might clean up application state which the message callback will attempt to use. This could lead to unexpected behavior, possibly including segmentation faults.

In this case, proper sequencing of operations is critical. The timer should delete the receiver first. While inside the receiver delete API, the XSP might deliver messages to the application. However, once the receiver delete API returns, it is guaranteed that the XSP is finished making receiver callbacks.

Note that in this example case, if the message receive callback attempts to cancel the timer, the cancel API will return an error. This is because the timer has already expired and the execution of the callback has begun, and is inside the receiver delete API. The message receive callback needs to be able to handle this sequence, presumably by not re-scheduling the timer.

7.1.3 XSP Usage

This section provides simplified C code fragments that demonstrate some of the XSP-related API calls. For full examples of XSP usage, see `lbmrcvxsp.c` (for C) and `lbmrcvxsp.java` (for Java).

Note

Each XSP thread has its own Unicast Listener (request) port. You may need to expand the range **request_tcp_port_low (context)** - **request_tcp_port_high (context)**.

The common sequence of operations during application initialization is minimally shown below. In the code fragments below, error detection and handling are omitted for clarity.

1. Create a context attribute object and set the **transport_mapping_function (context)** option to point at the application's XSP mapping callback function using the structure **lbm_transport_mapping_func_t**.

```
lbm_context_attr_t *ctx_attr;
err = lbm_context_attr_create(&ctx_attr);

lbm_transport_mapping_func_t mapping_func;
mapping_func.mapping_func = app_xsp_mapper_callback;
mapping_func.clientd = NULL; /            * Can include app state pointer. */

err = lbm_context_attr_setopt(ctx_attr, "transport_mapping_function",
        &mapping_func, sizeof(mapping_func));
```

2. Create the context.

```
err = lbm_context_create(&ctx, ctx_attr, NULL, NULL);
err = lbm_context_attr_delete(ctx_attr); / * No longer needed. */
```

3. Create XSPs using **lbm_xsp_create()**. In this example, only a single XSP is created.

```
lbm_xsp_t *xsp; / * app_xsp_mapper_callback() needs this; see below. */
err = lbm_xsp_create(&xsp, ctx, NULL, NULL);
```

Note that the application can optionally pass in a context attribute object and an XSP attribute object. The context attribute is because XSP is implemented as a sort of reduced-function sub-context, and so it is possible to modify context options for the XSP. However, this is rarely needed since the default action is for the XSP to inherit all the configuration of the main context.

4. Create a receiver for topic "X".

```
lbm_topic_t *topic;
err = lbm_rcv_topic_lookup(&topic, ctx, "X", NULL);

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

Event queues may also be used with XSP-assigned Transport Sessions.

5. At this point, when the main context discovers a source for topic "X", it will proceed to join the Transport Session. It will call the application's app_xsp_mapper_callback() function, which is minimally this:

```
lbm_xsp_t *app_xsp_mapper_callback(lbm_context_t *ctx,
        lbm_new_transport_info_t *transp_info, void *clientd)
{/
  * Retrieve the XSP object created in step 3. */
  return xsp;
}
```

This minimal callback simply returns the XSP that was created during initialization (the "clientd" can be helpful for that). By assigning all receiver Transport Sessions to the same XSP, you have effectively separated message processing from UM housekeeping tasks, like processing of topic resolution and timers. This can greatly reduce latency outliers.

As described in XSP Handles Transport Sessions, Not Topics, some users want to have multiple XSPs and assign the Transport Sessions to XSPs according to application logic. Note that the passed-in **lbm_new_transport_info_t** structure contains information about the Transport Session, such as the IP address of the sender. However, this structure does not contain topic information. Applications can use the resolver's source notification callback via the **resolver_source_notification_function (context)** attribute option to associate topics with source strings.

Note

Most of the time, the application mapping callback will be invoked each time a Transport Session is joined. However, there is one exception to this rule. If a context is already joined to a Transport Session carried on a multicast group and destination port, joining another Transport Session on the same multicast group and destination port does not invoke the mapping callback again. This is because the same socket is used for all Transport Sessions that use the same group:port.

7.1.4 Other XSP Operations

As of UM 6.12, XSP supports persistent receivers.

When an XSP object is created, an XSP attribute object can be supplied to set XSP options. The XSP options are:

- **operational_mode (xsp)**
- **zero_transports_function (xsp)**

To create and manipulate an XSP attribute object, see:

- **lbm_xsp_attr_create()**

- **lbm_xsp_attr_setopt()**
- **lbm_xsp_attr_getopt()**
- **lbm_xsp_attr_delete()**

To delete an XSP, all receivers associated with Transport Sessions handled by that XSP must first be deleted. Then the XSP can be deleted using **lbm_xsp_delete()**.

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

- **lbm_xsp_register_fd()**
- **lbm_xsp_cancel_fd()**

7.1.5 XSP Limitations

There are some restrictions and limitations on the XSP feature.

- The only transport types currently supported are LBT-RM, LBT-RU, and TCP. IPC, SMX, DBL, and BROKER are not supported with XSPs at this time.
- The ULB feature is not currently supported.
- The use of XSP is not currently compatible with Hot Failover (HF). If you desire to use Hot Failover with XSP, contact Support.

7.2 Using Late Join

This section introduces the use of Ultra Messaging Late Join in default and specialized configurations. See **Late Join Options** for more information.

Note

> If your application is running within a Ultra Messaging context with configuration option **request_tcp_bind_request_port (context)** set to zero, then request port binding has been turned off, which also disables the Late Join feature.
> With the UMQ product, you cannot use Late Join with Queuing.

The Late Join feature enables newly created receivers to receive previously transmitted messages. Sources configured for Late Join maintain a retention buffer (not to be confused with a transport retransmission window), which holds transmitted messages for late-joining receivers.

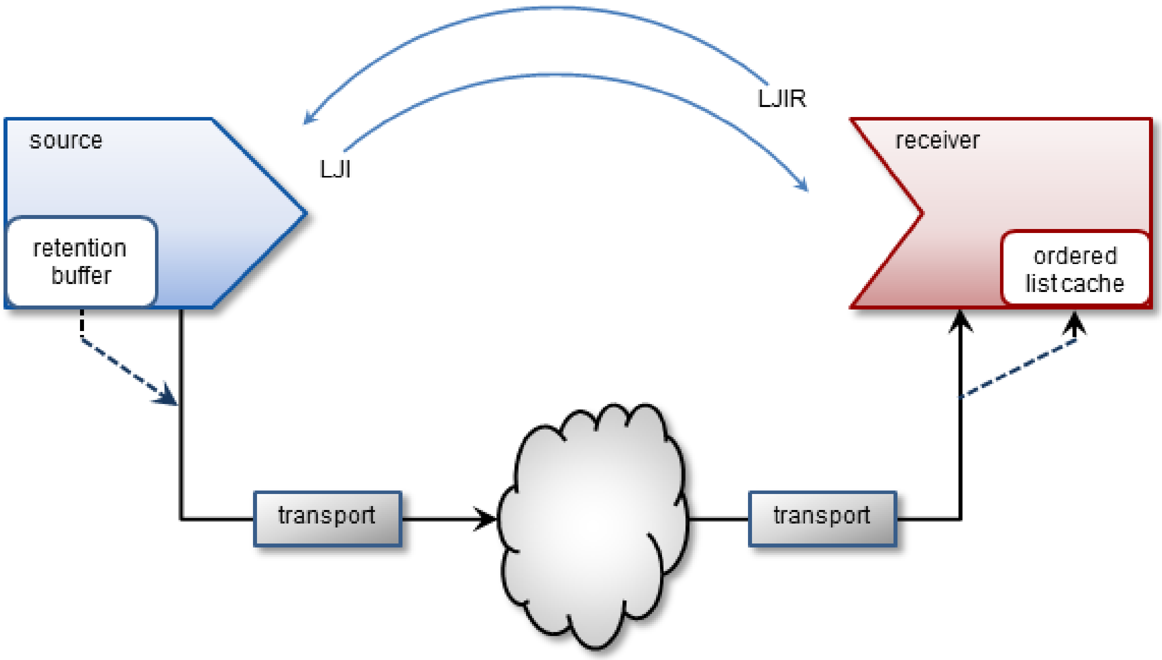
A Late Join operation follows the following sequence:

1. A new receiver configured for Late Join with **use_late_join (receiver)** completes topic resolution. Topic advertisements from the source contain a flag that indicates the source is configured for Late Join with **late_join (source)**.
2. The new receiver sends a Late Join Information Request (LJIR) to request a previously transmitted messages. The receiver configuration option, **retransmit_request_outstanding_maximum (receiver)**, determines the number of messages the receiver requests.

3. The source responds with a Late Join Information (LJI) message containing the sequence numbers for the retained messages that are available for retransmission.

4. The source unicasts the messages.

5. When Configuring Late Join for Large Numbers of Messages, the receiver issues additional requests, and the source retransmits these additional groups of older messages, oldest first.



The source's retention buffer's is not pre-allocated and occupies an increasing amount of memory as the source sends messages and adds them to the buffer. If a retention buffer grows to a size equal to the value of the source configuration option, **retransmit_retention_size_threshold (source)**, the source deletes older messages as it adds new ones. The source configuration option **retransmit_retention_age_threshold (source)**, controls message deletion based on message age.

UM uses control-structure overhead memory on a per-message basis for messages held in the retention buffer, in addition to the retention buffer's memory. Such memory usage can become significantly higher when retained messages are smaller in size, since more of them can then fit in the retention buffer.

Note

> If you set the receiver configuration option **ordered_delivery (receiver)** to 1, the receiver must deliver messages to your application in sequence number order. The receiver holds out-of-order messages in an ordered list cache until messages arrive to fill the sequence number gaps. If an out-of-order message arrives with a sequence number that creates a message gap greater than the value of **retransmit_message_caching_proximity (receiver)**, the receiver creates a burst loss event and terminates the Late Join recovery operation. You can increase the value of the proximity option and restart the receiver, but a burst loss is a significant event and you should investigate your network and message system components for failures.

7.2.1 Late Join With Persistence

With the UMP/UMQ products, late Join can be implemented in conjunction with the Persistent Store, however in this configuration, it functions somewhat differently from Streaming. After a late-Join-enabled receiver has been created, resolved a topic, and become registered with a store, it may then request older messages. The store unicasts the retransmission messages. If the store does not have these messages, it requests them of the source (assuming option retransmission-request-forwarding is enabled), thus initiating Late Join.

7.2.2 Late Join Options Summary

- **late_join (source)**
- **retransmit_retention_age_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_size_threshold (source)**
- **use_late_join (receiver)**
- **retransmit_initial_sequence_number_request (receiver)**
- **retransmit_message_caching_proximity (receiver)**
- **retransmit_request_message_timeout (receiver)**
- **retransmit_request_interval (receiver)**
- **retransmit_request_maximum (receiver)**
- **retransmit_request_outstanding_maximum (receiver)**

7.2.3 Using Default Late Join Options

To implement Late Join with default options, set the Late Join configuration options to activate the feature on both a source and receiver in the following manner.

1. Create a configuration file with source and receiver Late Join activation options set to 1. For example, file cfg1.cfg containing the two lines:

   ```
   source late_join 1
   receiver use_late_join 1
   ```

2. Run an application that starts a Late-Join-enabled source. For example:

   ```
   lbmsrc -c cfg1.cfg -P 1000 topicName
   ```

3. Wait a few seconds, then run an application that starts a Late-Join-enabled receiver. For example:

   ```
   lbmrcv -c cfg1.cfg -v topicName
   ```

The output for each should closely resemble the following:

LBMSRC

```
$ lbmsrc -c cfg1.cfg -P 1000 topicName
LOG Level 5: NOTICE: Source "topicName" has no retention settings (1 message
    retained max)
Sending 10000000 messages of size 25 bytes to topic [topicName]
Receiver connect [TCP:10.29.3.77:34200]
```

LBMRCV

```
$ lbmrcv -c cfg1.cfg -v topicName
Immediate messaging target: TCP:10.29.3.77:4391
[topicName][TCP:10.29.3.76:4371][2]-RX-, 25 bytes
1.001 secs. 0.0009988 Kmsgs/sec. 0.1998 Kbps
[topicName][TCP:10.29.3.76:4371][3], 25 bytes
1.002 secs. 0.0009982 Kmsgs/sec. 0.1996 Kbps
[topicName][TCP:10.29.3.76:4371][4], 25 bytes
1.003 secs. 0.0009972 Kmsgs/sec. 0.1994 Kbps
[topicName][TCP:10.29.3.76:4371][5], 25 bytes
1.003 secs. 0.0009972 Kmsgs/sec. 0.1994 Kbps
...
```

Note that the source only retained 1 Late Join message (due to default retention settings) and that this message appears as a retransmit (-RX-). Also note that it is possible to sometimes receive 2 RX messages in this scenario (see Retransmitting Only Recent Messages.)

7.2.4 Specifying a Range of Messages to Retransmit

To receive more than one or two Late Join messages, increase the source's **retransmit_retention_size_threshold (source)** from its default value of 0. Once the buffer exceeds this threshold, the source allows the next new message entering the retention buffer to bump out the oldest one. Note that this threshold's units are bytes (which includes a small overhead per message).

While the retention threshold endeavors to keep the buffer size close to its value, it does not set hard upper limit for retention buffer size. For this, the **retransmit_retention_size_limit (source)** configuration option (also in bytes) sets this boundary.

Follow the steps below to demonstrate how a source can retain about 50MB of messages, but no more than 60MB:

1. Create a second configuration file (cfg2.cfg) with the following options:

   ```
   source late_join 1
   source retransmit_retention_size_threshold 50000000
   source retransmit_retention_size_limit 60000000
   receiver use_late_join 1
   ```

2. Run lbmsrc -c cfg2.cfg -P 1000 topicName.

3. Wait a few seconds and run `lbmrcv -c cfg2.cfg -v topicName`. The output for each should closely resemble the following:

LBMSRC

```
$ lbmsrc -c cfg2.cfg -P 1000 topicName
Sending 10000000 messages of size 25 bytes to topic [topicName]
Receiver connect [TCP:10.29.3.76:34444]
```

LBMRCV

```
$ lbmrcv -c cfg2.cfg -v topicName
Immediate messaging target: TCP:10.29.3.76:4391
[topicName][TCP:10.29.3.77:4371][0]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][1]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][2]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][3]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][4]-RX-, 25 bytes
1.002 secs. 0.004991 Kmsgs/sec. 0.9981 Kbps
[topicName][TCP:10.29.3.77:4371][5], 25 bytes
```

```
1.002 secs. 0.0009984 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][6], 25 bytes
1.002 secs. 0.0009983 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][7], 25 bytes
...
```

Note that lbmrcv received live messages with sequence numbers 7, 6, and 5, and RX messages going from 4 all the way back to Sequence Number 0.

7.2.5 Retransmitting Only Recent Messages

Thus far we have worked with only source late join settings, but suppose that you want to receive only the last 10 messages. To do this, configure the receiver option **retransmit_request_maximum (receiver)** to set how many messages to request backwards from the latest message.

Follow the steps below to set this option to 10.

1. Add the following line to cfg2.cfg and rename it cfg3.cfg:

   ```
   receiver retransmitrequestmaximumreceiver 10
   ```

2. Run:

   ```
   lbmsrc -c cfg3.cfg -P 1000 topicName
   ```

3. Wait a few seconds and run `lbmrcv -c cfg3.cfg -v topicName`. The output for each should closely resemble the following.

LBMSRC

```
$ lbmsrc -c cfg3.cfg -P 1000 topicName
Sending 10000000 messages of size 25 bytes to topic [topicName]
Receiver connect [TCP:10.29.3.76:34448]
```

LBMRCV

```
$ lbmrcv -c cfg3.cfg -v topicName
Immediate messaging target: TCP:10.29.3.76:4391
[topicName][TCP:10.29.3.77:4371][13]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][14]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][15]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][16]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][17]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][18]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][19]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][20]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][21]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][22]-RX-, 25 bytes
[topicName][TCP:10.29.3.77:4371][23]-RX-, 25 bytes
1.002 secs. 0.01097 Kmsgs/sec. 2.195 Kbps
[topicName][TCP:10.29.3.77:4371][24], 25 bytes
1.002 secs. 0.0009984 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][25], 25 bytes
1.002 secs. 0.0009984 Kmsgs/sec. 0.1997 Kbps
[topicName][TCP:10.29.3.77:4371][26], 25 bytes
...
```

Note that 11, not 10, retransmits were actually received. This can happen because network and timing circumstances may have one RX already in transit while the specific RX amount is being processed. (Hence, it is not possible to guarantee one and only one RX message for every possible Late Join recovery.)

7.2.6 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.

Option: retransmit_request_outstanding_maximum (receiver)

When a receiver comes up and begins requesting Late Join messages, it does not simply request messages starting at Sequence Number 0 through 1000000. Rather, it requests the messages a little at a time, depending upon how option **retransmit_request_outstanding_maximum (receiver)** is set. For example, when set to the default of 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.

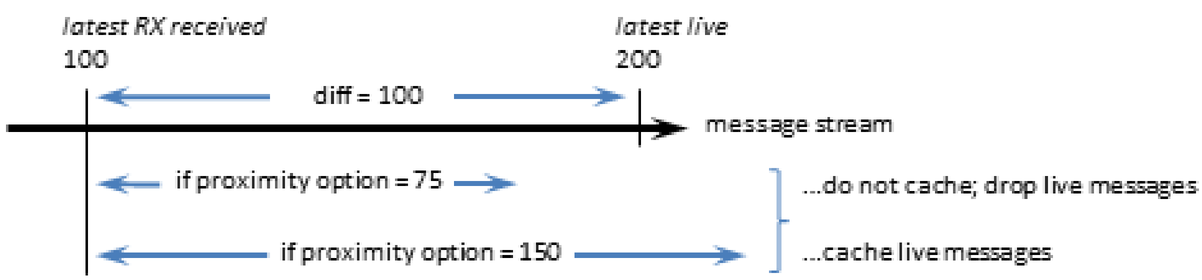
Note that in some environments, the default of 200 messages may be too high and overwhelm receivers with RXs, which can cause loss in a live LBT-RM stream. However, in other situations higher values can increase the rate of RXs received.

Option: retransmit_message_caching_proximity (receiver)

When sequence number delivery order is used, long recoveries of active sources can create receiver memory cache problems due to the processing of both new and retransmitted messages. This option provides a method to control caching and cache size during recovery.

It does this by comparing the option value (default 2147483647) to the difference between the newest (live) received sequence number and the latest received RX sequence number. If the difference is less than the option's value, the receiver caches incoming live new messages. Otherwise, new messages are dropped and not cached (with the assumption that they can be requested later as retransmissions).

For example, as shown in the diagram below, a receiver may be receiving both live streaming messages (latest, #200) and catch-up retransmissions (latest, #100). The difference here is 100. If **retransmit_message_caching↩ _proximity (receiver)** is 75, the receiver caches the live messages and will deliver them when it is all caught up with the retransmissions. However, if this option is 150, streamed messages are dropped and later picked up again as a retransmission.



The default value of this option is high enough to still encourage caching most of the time, and should be optimal for most receivers.

If your source streams faster than it retransmits, caching is beneficial, as it ensures new data are received only once, thus reducing recovery time. If the source retransmits faster than it streams, which is the optimal condition, you can lower the value of this option to use less memory during recovery, with little performance impact.

7.3 Off-Transport Recovery (OTR)

Off-Transport Recovery (OTR) is a lost-message-recovery feature that provides a level of hedging against the possibility of brief and incidental unrecoverable loss at the transport level or from a UM Router. This section describes the OTR feature.

Note

With the UMQ product, you cannot use OTR with Brokered Queuing.

When a transport cannot recover lost messages, OTR engages and looks to the source for message recovery. It does this by accessing the source's retention buffer (used also by the Late Join feature) to re-request messages that no longer exist in a transport's transmission window, or other places such as a Persistent Store or redundant source.

OTR functions in a manner very similar to that of Late Join, but differs mainly in that it activates in message loss situations rather than following the creation of a receiver, and shares only the **late_join (source)** option setting.

Upon detecting loss, a receiver initiates OTR by sending repeated, spaced, OTR requests to the source, until it recovers lost messages or a timeout period elapses.

OTR operates independently from transport-level recovery mechanisms such as NAKs for LBT-RU or LBT-RM. When you enable OTR for a receiver with **use_otr (receiver)**, the **otr_request_initial_delay (receiver)** period starts as soon as the Delivery Controller detects a sequence gap. If the gap is not resolved by the end of the delay interval, OTR recovery initiates. OTR recovery can occur before, during or after transport-level recovery attempts.

When a receiver initiates OTR, the intervals between OTR requests increases twofold after each request, until the maximum interval is reached (assuming the receiver is still waiting to receive the retransmission). You use configuration options **otr_request_minimum_interval (receiver)** and **otr_request_maximum_interval (receiver)** to set the initial (minimum) and maximum intervals, respectively.

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

7.3.1 OTR with Sequence Number Ordered Delivery

When sequence number delivery order is used and a gap of missing messages occurs, a receiver buffers the new incoming messages while it attempts to recover the earlier missing ones. Long recoveries of actively streaming sources can cause excessive receiver cache memory growth due to the processing of both new and retransmitted messages. You can control caching and cache size during recovery with options **otr_message_caching_threshold (receiver)** and **retransmit_message_caching_proximity (receiver)**.

The option **otr_message_caching_threshold (receiver)** sets the maximum number of messages a receiver can buffer. When the number of cached messages hits this threshold, new streamed messages are dropped and not cached, with the assumption that they can be requested later as retransmissions.

The **retransmit_message_caching_proximity (receiver)**, which is also used by Late Join (see **retransmit_message_caching_proximity (receiver)**), turns off this caching if there are too many messages to buffer between the last delivered message and the currently streaming messages.

Both of these option thresholds must be satisfied before caching resumes.

7.3.2 OTR With Persistence

With the UMP/UMQ products, you can implement OTR in conjunction with the Persistent Store, however in this configuration, it functions somewhat differently from Streaming. If an OTR-enabled receiver registered with a store detects a sequence gap in the live stream and that gap is not resolved by other means within the next **otr_request_initial_delay (receiver)** period, the receiver requests those messages from the store(s). If the store does not have some of the requested messages, the receiver requests them from the source. Regardless of whether the messages are recovered from a store or from the source, OTR delivers all recovered messages with the LBM_MSG_OTR flag, unlike Late Join, which uses the LBM_MSG_RETRANSMIT flag.

7.3.3 OTR Options Summary

- **late_join (source)**
- **retransmit_retention_age_threshold (source)**
- **retransmit_retention_size_limit (source)**
- **retransmit_retention_size_threshold (source)**
- **use_otr (receiver)**
- **otr_request_message_timeout (receiver)**
- **otr_request_initial_delay (receiver)**
- **otr_request_log_alert_cooldown (receiver)**
- **otr_request_maximum_interval (receiver)**
- **otr_request_minimum_interval (receiver)**
- **otr_request_outstanding_maximum (receiver)**
- **otr_message_caching_threshold (receiver)**
- **retransmit_message_caching_proximity (receiver)**

Note

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

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

7.4 Encrypted TCP

This section introduces the use of Transport Layer Security (TLS), sometimes known by its older designation Secure Sockets Layer (SSL).

The goal of the Ultra Messaging (UM) TLS feature is to provide encrypted transport of application data. TLS supports authentication (through certificates), data confidentiality (through encryption), and data integrity (ensuring data are not changed, removed, or added-to). UM can be configured to apply TLS security measures to all Streaming and/or Persisted TCP communication, including UM Router peer links. Non-TCP communication is not encrypted (e.g. topic resolution).

TLS is a family of standard protocols and algorithms for securing TCP communication between a client and a server. It is sometimes referred as "SSL", which technically is the name of an older (less secure) version of the protocol.

Over the years, security researchers (and hackers) have discovered flaws in SSL/TLS. However, the vast majority of the widely publicized security vulnerabilities have been flaws in the implementations of TLS, not in the recent TLS protocols or algorithms themselves. As of UM version 6.9, there are no known security weaknesses in TLS version 1.2, the version used by UM.

TLS is generally implemented by several different software packages. UM makes use of OpenSSL, a widely deployed and actively maintained open-source project.

7.4.1 TLS Authentication

TLS authentication uses X.509 digital certificates. Certificate creation and management is the responsibility of the user. Ultra Messaging's usage of OpenSSL expects PEM encoded certificates. There are a variety of generally available tools for converting certificates between different encodings. Since user infrastructures vary widely, the UM package does not include tools for creation, formatting, or management of certificates.

Although UM is designed as a peer-to-peer messaging system, TLS has the concept of client and server. The client initiates the TCP connection and the server accepts it. In the case of a TCP source, the receiver initiates and is therefore the client, with the source (sender of data) being the server. However, with unicast immediate messages, the sender of data is the client, and the recipient is the server. Due to the fact that unicast immediate messages are used by UM for internal control and coordination, it is typically not possible to constrain a given application to only operate as a pure client or pure server. For this reason, UM requires all applications participating in encryption to have a certificate. Server-only authentication (i.e. anonymous client, as is used by web browsers) is not supported. It is permissible for groups of processes, or even all processes, to share the same certificate.

A detailed discussion of certificate usage is beyond the scope of the Ultra Messaging documentation.

7.4.2 TLS Backwards Compatibility

The TLS protocol was designed to allow for a high degree of backwards compatibility. During the connection establishment phase, the client and server perform a negotiation handshake in which they identify the highest common versions of various security options. For example, an old web browser might pre-date the introduction of TLS and only support the older SSL protocol. OpenSSL is often configured to allow clients and servers to "negotiate down" to those older, less-secure protocols or algorithms.

Ultra Messaging has the advantage of not needing to communicate with old versions of SSL or TLS. UM's default configuration directs OpenSSL to require both the client and the server to use protocols and algorithms which were highly regarded, as of UM's release date. If vulnerabilities are discovered in the future, the user can override UM's defaults and chose other protocols or algorithms.

7.4.3 TLS Efficiency

When a TLS connection is initiated, a handshake takes place prior to application data encryption. Once the handshake is completed, the CPU effort required to encrypt and decrypt application data is minimal. However, the handshake phase involves the use of much less efficient algorithms.

There are two factors under the user's control, which greatly affect the handshake efficiency: the choice of cipher suite and the key length. We have seen an RSA key of 8192 bits take 4 seconds of CPU time on a 1.3GHz SparcV9 processor just to complete the handshake for a single TLS connection.

Users should make their choices with an understanding of the threat profiles they are protecting against. For example, it is estimated that a 1024-bit RSA key can be broken in about a year by brute force using specialized hardware (see http://www.tau.ac.il/~tromer/papers/cbtwirl.pdf). This may be beyond the means of the average hacker, but well within the means of a large government. RSA keys of 2048 bits are generally considered secure for the foreseeable future.

7.4.4 TLS Configuration

TLS is enabled on a context basis. When enabled, all Streaming and Persistence related TCP-based communication into or out of the context is encrypted by TLS. A context with TLS enabled will not accept source creation with transports other than TCP.

Subscribers will only successfully receive data if the receiver's context and the source's context share the same encryption settings. A receiver created in an encrypted enabled context will ignore topic resolution source advertisements for non-encrypted sources, and will therefore not subscribe. Similarly, a receiver created in a non-encrypted context will ignore topic resolution source advertisements for encrypted sources. Topic resolution queries are also ignored by mismatched contexts. No warning will be logged when these topic resolution datagrams are ignored, but each time this happens, the context-level statistic tr_dgrams_dropped_type is incremented.

TLS is applied to unicast immediate messages as well, as invoked either directly by the user, or internally by functions like late join, request/response, and Persistence-related communication between sources, receivers, and stores.

Brokered Queuing using AMQP does not use the UM TLS feature. A UM brokered context does not allow TLS to be enabled.

7.4.5 TLS Options Summary

- **use_tls (context)**
- **tls_cipher_suites (context)**
- **tls_certificate (context)**
- **tls_certificate_key (context)**
- **tls_certificate_key_password (context)**
- **tls_trusted_certificates (context)**
- **tls_compression_negotiation_timeout (context)**

The **tls_cipher_suites (context)** configuration option defines the list of one or more (comma separated) cipher suites that are acceptable to this context. If more than one is supplied, they should be in descending order of preference. When a remote context negotiates encrypted TCP, the two sides must find a cipher suite in common, otherwise the connection will be canceled.

OpenSSL uses the cipher suite to define the algorithms and key lengths for encrypting the data stream. The choice of cipher suite is critical for ensuring the security of the connection. To achieve a high degree of backwards compatibility, OpenSSL supports old cipher suites which are no longer considered secure. The user is advised to use UM's default suite.

OpenSSL follows its own naming convention for cipher suites. See https://www.openssl.org/docs/manmaster/apps/ciphers.html#TLS-v1.2-cipher-suites for a list of valid suite names (the ones with dashes) and the equivalent IANA names (with underscores). The UM configuration should use the OpenSSL-style names (with dashes).

7.4.6 TLS and Persistence

TLS is designed to encrypt a TCP connection, and works with TCP-based persisted data Transport Sessions and control traffic. However, TLS is not intended to encrypt data at rest. When a Persistent Store is used with the UM TLS feature, the user messages are written to disk in plaintext form, not encrypted.

7.4.7 TLS and Queuing

The UM TLS feature does not apply to the AMQP connection to the brokered queue. UM does not currently support security on the AMQP connection.

However, the ULB form of queuing does not use a broker. For ULB sources that are configured for TCP, the UM TLS feature will encrypt the application data.

7.4.8 TLS and the Dynamic Routing Option (DRO)

When a UM Router is used to route messages across Topic Resolution Domains (TRDs), be aware that the TLS session is terminated at the UM Router's proxy receiver/source. Because each endpoint portal on a UM Router is implemented with its own context, care must be taken to ensure end-to-end security. It is possible to have a TLS source publishing in one TRD, received by a UM Router (via an endpoint portal also configured for TLS), and re-published to a different TRD via an endpoint portal configured with a non-encrypted context. This would allow a non-encrypted receiver to access messages that the source intended to be encrypted. As a message is forwarded through a UM Router network, it does not propagate the security settings of the originator, so each portal needs to be appropriately encrypted. The user is strongly encouraged to configure ALL portals on an interconnected network of UM Routers with the same encryption settings.

The encryption feature is extended to UM Router peer links, however peer links are not context-based and are not configured the same way. The following XML elements are used by the UM Router to configure a peer link:

- '<**tls**>'
- '<**cipher-suites**>'
- '<**certificate**>'
- '<**certificate-key**>'
- '<**certificate-key-password**>'
- '<**trusted-certificates**>'

As with sources and receivers, the portals on both sides of a peer link must be configured for compatible encryption settings.

Notice that there is no element corresponding to the context option **tls_compression_negotiation_timeout (context)**. The UM Router peer link's negotiation timeout is hard-coded to 5 seconds.

See the UM Router configuration DTD for details.

7.4.9 TLS and Compression

Many users have advanced network equipment (switches/routers), which transparently compress packets as they traverse the network. This compression is especially valued to conserve bandwidth over long-haul WAN links. However, when packets are encrypted, the network compressors are typically not able to reduce the size of the data. If the user desires UM messages to be compressed and encrypted, the data needs to be compressed before it is encrypted.

The UM compression feature (see Compressed TCP) accomplishes this. When both TLS and compression are enabled, the compression is applied to user data first, then encryption.

Be aware that there can be information leakage when compression is applied and an attacker is able to inject data of known content over a compressed and encrypted session. For example, this leakage is exploited by the `CRIME` attack, albeit primarily for web browsers. Users must weigh the benefits of compression against the potential risk of information leakage.

Version Interoperability

It is not recommended to mix pre-6.9 contexts with encrypted contexts on topics of shared interest. If a process with a pre-6.9 version of UM creates a receiver, and another process with UM 6.9 or beyond creates a TLS source, the pre-6.9 receiver will attempt to join the TLS source. After a timeout, the handshake will fail and the source will disconnect. The pre-6.9 receiver will retry the connection, leading to flapping.

Note that in the reverse situation, a 6.9 TLS receiver will simply ignore a pre-6.9 source. I.e. no attempt will be made to join, and no flapping will occur.

7.5 Compressed TCP

This section introduces the use of Compression with TCP connections.

The goal of the Ultra Messaging (UM) compression feature is to decrease the size of transmitted application data. UM can be configured to apply compression to all Streaming and/or Persisted TCP communication.

Non-TCP communication is not compressed (e.g. topic resolution).

Compression is generally implemented by any of several different software packages. UM makes use of LZ4, a widely deployed open-source project.

While the UM compression feature is usable for TCP-based sources and receivers, it is possibly most useful when applied to UM Router peer links.

7.5.1 Compression Configuration

Compression is enabled on a context basis. When enabled, all Streaming and Persistence related TCP-based communication into or out of the context is compressed by LZ4. A context with compression enabled will not accept source creation with transports other than TCP.

Subscribers will only successfully receive data if the receiver's context and the source's context share the same compression settings. A receiver created in a compression-enabled context will ignore topic resolution source advertisements for non-compressed sources, and will therefore not subscribe. Similarly, a receiver created in an non-compressed context will ignore topic resolution source advertisements for compressed sources. Topic resolution queries are also ignored by mismatched contexts. No warning will be logged when these topic resolution datagrams are ignored, but each time this happens, the context-level statistic tr_dgrams_dropped_type is incremented.

Compression is applied to unicast immediate messages as well, as invoked either directly by the user, or internally by functions like late join, request/response, and Persistence-related communication between sources, receivers, and stores.

Brokered Queuing using AMQP does not use the UM compression feature. A UM brokered context does not allow compression to be enabled.

The compression-related configuration options used by the Ultra Messaging library are:

- **compression (context)**
- **tls_compression_negotiation_timeout (context)**

7.5.2 Compression and Persistence

Compression is designed to compress a data Transport Session. It is not intended to compress data at rest. When a Persistent Store is used with the UM compression feature, the user messages are written to disk in uncompressed form.

7.5.3 Compression and Queuing

The UM compression feature does not apply to the AMQP connection to the brokered queue. UM does not currently support compression on the AMQP connection.

However, the ULB form of queuing does not use a broker. For ULB sources that are configured for TCP, the UM compression feature will compress the application data.

7.5.4 Compression and the Dynamic Routing Option (DRO)

When a UM Router is used to route messages across Topic Resolution Domains (TRDs), be aware that the compression session is terminated at the UM Router's proxy receiver/source. Because each endpoint portal on a UM Router is implemented with its own context, care must be taken to ensure end-to-end compression (if desired). As a message is forwarded through a UM Router network, it does not propagate the compression setting of the originator, so each portal needs to be appropriately compressed.

Possibly the most-useful application of the UM compression feature is not TCP sources, but rather UM Router peer links. The compression feature is extended to UM Router peer links, however peer links are not context-based and are not configured the same way. The following XML elements are used by the UM Router to configure a peer link:

- '<**compression**>'

As with sources and receivers, the portals on both sides of a peer link must be configured for the same compression setting.

Notice that there is no element corresponding to the context option **tls_compression_negotiation_timeout (context)**. The UM Router peer link's negotiation timeout is hard-coded to 5 seconds.

See the UM Router configuration DTD for details.

7.5.5 Compression and Encryption

See TLS and Compression.

7.5.6 Version Interoperability

It is not recommended to mix pre-6.9 contexts with compressed contexts on topics of shared interest. As mentioned above, if a compressed and an uncompressed context connect via TCP, the connection will fail and retry, resulting in flapping.

7.6 High-resolution Timestamps

This section introduces the use of high-resolution timestamps with LBT-RM.

The Ultra Messaging (UM) high-resolution message timestamp feature leverages the hardware timestamping function of certain Solarflare network interface cards (NICs) to measure sub-microsecond times that packets are transmitted and received. Solarflare's NICs and Onload kernel-bypass driver implement PTP to synchronize timestamps across the network, allowing very accurate one-way latency measurements. The UM timestamp feature requires Solarflare OpenOnload version 201509 or later.

For subscribers, each message's receive timestamp is delivered in the message's header structure (for C programs, **lbm_msg_t** field **hr_timestamp**, of type **lbm_timespec_t**). Each timestamp is a structure of 32 bits worth of seconds and 32 bits worth of nanoseconds. When both values are zero, the timestamp is not available.

For publishers, each message's transmit timestamp is delivered via the source event callback (for C programs, event type LBM_SRC_EVENT_TIMESTAMP). The same timestamp structure as above is delivered with the event, as well as the message's sequence number. Sending applications can be informed of the outgoing sequence number range of each message by using the extended form of the send function and supplying the LBM_SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO flag. This causes the LBM_SRC_EVENT_SEQUENCE_NUMBER_INFO event to be delivered to the source event handler.

7.6.1 Timestamp Restrictions

Due to the specialized nature of this feature, there are several restrictions in its use.

- **Operating system**: Linux only. No timestamps will be delivered on other operating systems. Also, since the feature makes use of the rcvmmsg() function, no timestamps will be delivered on Linux kernels prior to 2.6.33 and glibc libraries prior to 2.12 (which was released in 2010).
- **Languages**: C and Java only.
- **Transport**: Source-based LBT-RM (multicast) Transport Sessions only. No timestamps will be delivered for MIM or other transport types.
- **Queuing**: Timestamps are not supported for broker-based queuing. If a ULB source is configured for LBT-RM, send-side timestamps are not supported and will not be delivered if one or more receivers are registered. However, on the receive side, ULB messages are time stamped.
- **Loss**: If packet loss triggers LBT-RM's NAK/retransmit sequence, the send side will have multiple timestamps delivered, one for each multicast transmission. On the receive side, the timestamp of the first successfully received multicast datagram will be delivered.

- **Recovery**: For missed messages which are recovered via Late Join, Off-Transport Recovery (OTR), or the Persistent Store, no timestamp will be delivered, either on the send side or the receive side.
- **Implicit batching**: If implicit batching is being used, only the first message in a batch will have a send-side timestamp delivered. When implicit batching is used, the sender must be prepared for some messages to not have timestamps delivered. On the receive side, all messages in a batch will have the same timestamp.
- **UM Fragmentation, send-side**: If user messages are too large to be contained in a single datagram, UM will fragment the message into multiple datagrams. On the send side, each datagram will trigger delivery of a timestamp.
  - UM Fragmentation, receive-side with **ordered_delivery (receiver)** set to 0 (arrival): Arrival-order delivery will result in each fragment being delivered separately, as it is received. Each fragment's message header will contain a timestamp. Arrival order delivery provides an accurate timestamp of when the complete message is received (although, as mentioned above, any fragment recovered via OTR or the Persistent Store will not have a timestamp).
  - UM Fragmentation, receive-side with **ordered_delivery (receiver)** set to 1 or -1, (reassembly): Delivery with reassembly results in a single timestamp included in the message header. That timestamp corresponds to the arrival of the last fragment of the message (although, as mentioned above, any fragment recovered via OTR or the Persistent Store will not have a timestamp). Note that this is not necessarily the last fragment received; if an intermediate datagram is lost and subsequently re-transmitted after a delay, that intermediate datagram will be the last one received, but its timestamp will not be used for the message. For example, if a three-fragment message is received in the order of F0, F2, F1, the timestamp for the message will correspond to F2, the last fragment of the message. If fragmented messages are being sent, and an accurate time of message completion is needed, arrival order delivery must be used.
- **UM Fragmentation plus implicit batching**: If user messages vary widely in size, some requiring fragmentation, and implicit batching is used be aware that a full fragment does not completely fill a datagram. For example, if a small message (less than 300 bytes) is sent followed by a large message requiring fragmentation, the first fragment of the large message will fit in the same datagram as the small message. In that case, on the send side, a timestamp will not be delivered for that first fragment. However, a timestamp will be delivered for the second fragment. On the receive side, the same restrictions apply as described with UM fragmentation.
- **Local loopback**: If an LBT-RM source and receiver share the same physical machine, the receive side will not have timestamps delivered.

7.6.2 Timestamp Configuration Summary

- **transport_lbtrm_source_timestamp (context)**
- **transport_lbtrm_receiver_timestamp (context)**

7.7 Message Properties

The message property object **lbm_msg_properties_t** 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 set the desired message properties using **lbm_msg_properties_set()**. 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. For an example on how to iterate received message properties, see `lbmrcv.c`.

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. See Zero Object Delivery.

Note

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

With the UMQ product, the UM message property object supports the standard JMS message properties specification.

7.7.1 Smart Sources and Message Properties

Smart Sources support a limited form of message properties. Only 32-bit integer property types are allowed with Smart Sources. Also, property names are limited to 7 ASCII characters. Finally, the normal message properties object **lbm_msg_properties_t** and its APIs *are not used* on the sending side. Rather a streamlined method of specifying message properties for sending is used.

As with most of Smart Source's internal design, the message header for message properties must be pre-allocated with the maximum number of desired message properties. This is done at creation time for the Smart Source using the configuration option **smart_src_message_property_int_count (source)**.

Sending messages with message properties must be done using the **lbm_ssrc_send_ex()** API, passing it the desired properties with the **lbm_ssrc_send_ex_info_t** structure. The first call to send with message properties will serialize the supplied properties and encode them into the pre-allocated message header.

Subsequent calls to send with message properties will ignore the passed-in properties and simply re-send the previously-serialized header.

If an application needs to change the message property values after that initial send, the "update" flag flag can be used, which will trigger modification of the property values. This "update" flag cannot be used to change the number of properties, or the key names of the properties.

If an application needs messages with different numbers of properties and/or different key names of properties, the most efficient way to accomplish this is with multiple message buffers. Each buffer should be associated with a desired set of properties. When a message needs to be sent, the proper buffer is selected for building the message. This avoid the overhead of serializing the properties with each send call.

However, if the application requires dynamic construction of properties, a single buffer can be used along with the "rebuild" flag to trigger a full serialization of the properties.

Note

> If using both message properties and Spectrum with a single Smart Source, there is an added restriction: it is not possible to send a message omitting only one of those features. I.e. if both are enabled when the Smart Source is created, it is not possible to send a message with a message property and not a channel, and it is

not possible to send a message with a channel and not a property. This is because the message header is defined at Smart Source creation, and the header either must contain both or neither.

7.7.2 Smart Source Message Properties Usage

For a full example of message property usage with Smart Source, see lbmssrc.c or lbmssrc.java.

The first message with a message property sent to a Smart Source follows a specific sequence:

1. Create the topic object with the configuration option **smart_src_message_property_int_count (source)** set to the maximum number of properties desired on a message.

2. Create the Smart Source with **lbm_ssrc_create()**.

3. Allocate one or more message buffers with **lbm_ssrc_buff_get()**. You might allocate one for messages that have properties, and another for messages that don't.

4. When preparing the first message with message properties to be sent, define the properties using a **lbm_ssrc_send_ex_info_t** structure:

```
char *prop_name_array[3]; /* Array of property names. */
prop_name_array[0] = "abc"; /* 7 ascii characters or less. */
prop_name_array[1] = "XYZ";
prop_name_array[2] = "123";

lbm_int32_t prop_value_array[3]; /* Array of property values. */
prop_value_array[0] = 29;
prop_value_array[1] = -300;
prop_value_array[2] = 0;

lbm_ssrc_send_ex_info_t ss_send_info;
memset((char *)&ss_send_info, 0, sizeof(ss_send_info));
ss_send_info.mprop_int_cnt = 3;
ss_send_info.mprop_int_keys = prop_name_array;
ss_send_info.mprop_int_vals = prop_value_array;
```

5. Send the message using **lbm_ssrc_send_ex()** and the **LBM_SSRC_SEND_EX_FLAG_PROPERTIES** flag:

```
ss_send_info.flags = LBM_SSRC_SEND_EX_FLAG_PROPERTIES;
err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
```

   Since this is the first send with message properties, UM will serialize the properties and set up the message header. (It is not valid to set the **LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES** flag on this first send with message properties.)

For subsequent sends, there are different use cases:

- Send the message with the same properties and values. You can re-use the same message buffer and **lbm_ssrc_send_ex_info_t** structure:

```
/
  * The ss_send_info.flags still has LBM_SSRC_SEND_EX_FLAG_PROPERTIES set */
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
```

- Send with message properties after having made changes to the property values (but not the keys or the number of properties) by setting the **LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES** flag:

```
prop_value_array[0] = 28; / * Change property value. */
ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES;
err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
ss_send_info.flags &= ~LBM_SSRC_SEND_EX_FLAG_UPDATE_PROPERTY_VALUES;
```

- Send a message with either a different number of properties, and/or different key names by setting the **LB↩ M_SSRC_SEND_EX_FLAG_REBUILD_BUFFER** flag:

```
/
  * Send with only the first 2 properties. */
  ss_send_info.mprop_int_cnt = 2;
  ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
  ss_send_info.flags &= ~ LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
```

- Send a message without any message properties. This is a subset of the previous case (changing the number of properties). Use the **LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER** flag and clear the **LBM_SSR↩ C_SEND_EX_FLAG_PROPERTIES** flag:

```
/
  * Clear the properties flag so no properties will be sent. */
  ss_send_info.flags &= ~ LBM_SSRC_SEND_EX_FLAG_PROPERTIES;
  ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
  err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
  ss_send_info.flags &= ~ LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER;
```

To be more efficient, instead of using the **LBM_SSRC_SEND_EX_FLAG_REBUILD_BUFFER** flag, you can use different message buffers (allocated with **lbm_ssrc_buff_get()**) for each message property structure. This saves the time required to re-serialize the message properties each time you want to use a different property structure.

7.8 Request/Response Model

Request/response is a very common messaging model whereby a client sends a "request" message to a server and expects a response. The server processes the request and return a response message to the originating client.

The UM request/response feature simplifies implementation of this model in the following ways:

- Handling the request's "return address", eliminating the need for the client to create an artificial guaranteed-unique topic for the response.
- Establishing a linkage between a request and its response(s), allowing multiple requests to be outstanding, and associating each response message with its corresponding request message.
- Supporting multiple responses per request, both by allowing multiple servers to receive the request and each one responding, and by allowing a given server to respond with multiple messages.

7.8.1 Request Message

UM provides three ways to send a request message.

- **lbm_send_request()** to send a request to a topic via a source object. Uses the standard source-based transports (TCP, LBT-RM, LBT-RU).

- **lbm_multicast_immediate_request()** to send a request to a topic as a multicast immediate message. See Multicast Immediate Messaging.
- **lbm_unicast_immediate_request()** to send a request to a topic as a unicast immediate message.

When the client application sends a request message, it references an application callback function for responses and a client data pointer for application state. The send call returns a "request object". As one or more responses are returned, the callback is invoked to deliver the response messages, associated with the request's client data pointer. The requesting application decides when its request is satisfied (perhaps by completeness of a response, or by timeout), and it calls **lbm_request_delete()** to delete the request object. Even if the server chooses to send additional responses, they will not be delivered to the requesting application after it has deleted the corresponding request object.

7.8.2 Response Message

The server application receives a request via the normal message receive mechanism, but the message is identified as type "request". Contained within that request message's header is a response object, which serves as a return address to the requester. The server application responds to an UM request message by calling **lbm_send_response()**. The response message is sent unicast via a dynamic TCP connection managed by UM.

Warning

> The **lbm_send_response()** function may not be called from a context thread callback. If the application needs to send the response from the receiver callback, it must associate that receiver callback with an event queue.

Note

> Since the response object is part of the message header, it is normally deleted at the same time that the message is deleted, which typically happens automatically when the receiver callback returns. However, there are times when the application needs the scope of the response object to extend beyond the execution of the receiver callback. One method of extending the lifetime of the response object is to "retain" the request message, using **lbm_msg_retain()**.
>
> However, there are times when the size of the request message makes retention of the entire message undesirable. In those cases, the response object itself can be extracted and retained separately by saving a copy of the response object pointer and setting the message header's response pointer to NULL (to prevent UM from deleting the response object when the message is deleted).
>
> There are even occasions when an application needs to transfer the responsibility of responding to a request message to a different process entirely. I.e. the server which receives the request is not itself able to respond, and needs to send a message (not necessarily the original request message) to a different server. In that case, the first server which receives the request must serialize the response object to type **lbm_serialized_response_t** by calling **lbm_serialize_response()**. It includes the serialized response object in the message forwarded to the second server. That server de-serializes the response object by calling **lbm_deserialize_response()**, allowing it to send a response message to the original requesting client.

7.8.3 TCP Management

UM creates and manages the special TCP connections for responses, maintaining a list of active response connections. When an application sends a response, UM scans that list for an active connection to the destination. If it doesn't find a connection for the response, it creates a new connection and adds it to the list. After the **lbm_send_response()** function returns, UM schedules the **response_tcp_deletion_timeout (context)**, which defaults to 2 seconds. If a second request comes in from the same application before the timer expires, the responding application simply uses the existing connection and restarts the deletion timer.

It is conceivable that a very large response could take more than the **response_tcp_deletion_timeout (context)** default (2 seconds) to send to a slow-running receiver. In this case, UM automatically increases the deletion timer as needed to ensure the last message completes.

7.8.4 Request/Response Configuration

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

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

Note

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

7.8.5 Request/Response 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:

- Run **lbmresp -v topicname**
- Run **lbmreq -R 5 -v topicname**

LBMREQ

Output for lbmreq should resemble the following:

```
$ lbmreq -R 5 -q topicname
Event queue in use
Using TCP port 4392 for responses
Delaying requests for 1000 milliseconds
Sending request 0
Starting event pump for 5 seconds.
Receiver connect [TCP:10.29.1.78:4958]
```

```
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 1
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 2
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 3
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
    Sending request 4
Starting event pump for 5 seconds.
Done waiting for responses. 1 responses (25 bytes) received. Deleting request.
Quitting...
```

LBMRESP

Output for lbmresp should resemble the following:

```
$ lbmresp -v topicname
Request [topicname][TCP:10.29.1.78:14371][0], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][1], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][2], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][3], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
Request [topicname][TCP:10.29.1.78:14371][4], 25 bytes
Sending response. 1 responses of 25 bytes each (25 total bytes).
Done sending responses. Deleting response.
[topicname][TCP:10.29.1.78:14371], End of Transport Session
```

7.9 Self Describing Messaging

The UM Self-Describing Messaging (SDM) feature provides an API that simplifies the creation and use of messages by your applications. An SDM message contains one or more fields and each field consists of the following:

- A name
- A type
- A value

Each named field may appear only once in a message. If multiple fields of the same name and type are needed, array fields are available. A field in a nested message may have the same name as a field in the outer message.

SDM is particularly helpful for creating messages sent across platforms by simplifying the creation of data formats. SDM automatically performs platform-specific data translations, eliminating endian conflicts.

Using SDM also simplifies message maintenance because the message format or structure can be independent of the source and receiver applications. For example, if your receivers query SDM messages for particular fields and ignore the order of the fields within the message, a source can change the field order if necessary with no modification of the receivers needed.

See the C, Java, and .NET API guides for details.

7.10 Pre-Defined Messages

The UM Pre-Defined Messages (PDM) feature provides an API similar to the SDM API, but allows you to define messages once and then use the definition to create messages that may contain self-describing data. Eliminating the need to repeatedly send a message definition increases the speed of PDM over SDM. The ability to use arrays created in a different programming language also improves performance.

The PDM library lets you create, serialize, and deserialize messages using pre-defined knowledge about the possible fields that may be used. You can create a definition that a) describes the fields to be sent and received in a message, b) creates the corresponding message, and c) adds field values to the message. This approach offers several performance advantages over SDM, as the definition is known in advance. However, the usage pattern is slightly different 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.

7.10.1 Typical PDM Usage Patterns

The typical PDM usage patterns can usually be broken down into two categories: sources (which need to serialize a message for sending) and receivers (which need to deserialize a message to extract field values). However, for optimum performance for both sources and receivers, first set up the definition and a single instance of the message only once during a setup or initialization phase, as in the following example workflow:

1. Create a definition and set its id and version.
2. Add field information to the definition to describe the types of fields to be in the message.
3. Create a single instance of a message based on the definition.
4. Set up a source to do the following:
   - Add field values to the message instance.
   - Serialize the message so that it can be sent.
5. Likewise, set up a receiver to do the following:
   - Deserialize the received bytes into the message instance.
   - Extract the field values from the message.

7.10.2 Getting Started with PDM

PDM APIs are provided in C, Java, and C#, however, the examples in this section are Java based.

PDM Code Example, Source

Translating the Typical PDM Usage Patterns to Java for a source produces the following:

```
private PDMDefinition defn;
private PDMMessage msg;
private PDMFieldInfo fldInfo100;
private PDMFieldInfo fldInfo101;
private PDMFieldInfo fldInfo102;

public void setupPDM() {//
  Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);//

  Set the definition id and version
  defn.setId(1001);
  defn.setMsgVersMajor((byte)1);
  defn.setMsgVersMinor((byte)0);//

  Create information for a boolean, int32, and float fields (all required)
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);//

  Finalize the definition and create the message defn.finalizeDef();
  msg = new PDMMessage(defn);
}

public void sourceUsePDM() {//
  Call the function to setup the definition and message
  setupPDM();//

  Example values for the message boolean
  fld100Val = true;
  int fld101Val = 7;
  float fld102Val = 3.14F;//

  Set each field value in the message
  msg.setFieldValue(fldInfo100, fld100Val);
  msg.setFieldValue(fldInfo101, fld101Val);
  msg.setFieldValue(fldInfo102, fld102Val);//

  Serialize the message to bytes
  byte[] buffer = msg.toBytes();
}
```

PDM Code Example, Receiver

Translating the Typical PDM Usage Patterns to Java for a receiver produces the following:

```
private PDMDefinition defn;
private PDMMessage msg;
private PDMFieldInfo fldInfo100;
private PDMFieldInfo fldInfo101;
private PDMFieldInfo fldInfo102;

public void setupPDM() {//
  Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);//

  Set the definition id and version
  defn.setId(1001);
```

```
  defn.setMsgVersMajor((byte)1);
  defn.setMsgVersMinor((byte)0);//

  Create information for a boolean, int32, and float field (all required)
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);//

  Finalize the definition and create the message
  defn.finalizeDef();
  msg = new PDMMessage(defn);
}

public void receiverUsePDM(byte[] buffer) {//
  Call the function to setup the definition and message
  setupPDM();//

  Values to be retrieved from the message
  boolean fld100Val;
  int fld101Val;
  float fld102Val;//

  Deserialize the bytes into a message
  msg.parse(buffer);//

  Get each field value from the message
  fld100Val = msg.getFieldValueAsBoolean(fldInfo100);
  fld101Val = msg.getFieldValueAsInt32(fldInfo101);
  fld102Val = msg.getFieldValueAsFloat(fldInfo102);
}
```

PDM Code Example Notes

In the examples above, the setupPDM() function is called once to set up the PDM definition and message. It is identical in both the source and receiver cases and simply sets up a definition that contains three required fields with integer names (100, 101, 102). Once finalized, it can create a message that leverages its pre-defined knowledge about these three required fields. The source example adds the three sample field values (a boolean, int32, and float) to the message, which is then serialized to a byte array. In the receiver example, the message parses a byte array into the message and then extracts the three field values.

7.10.3 Using the PDM API

The following code snippets expand upon the previous examples to demonstrate the usage of additional PDM functionality (but use "..." to eliminate redundant code).

Reusing the Message Object

Although the examples use a single message object (which provides performance benefits due to reduced message creation and garbage collection), it is not explicitly required to reuse a single instance. However, multiple threads should not access a single message instance.

Number of Fields

Although the number of fields above is initially set to 3 in the PDMDefinition constructor, if you add more fields to the definition with the addFieldInfo method, the definition grows to accommodate each field. Once the definition is finalized, you cannot add additional field information because the definition is now locked and ready for use in a message.

String Field Names

The examples above use integer field names in the setupPDM() function when creating the definition. You can also use string field names when setting up the definition. However, you still must use a FieldInfo object to set or get a field value from a message, regardless of field name type. Notice that false is passed to the PDMDefinition constructor to indicate string field names should be used. Also, the overloaded addFieldInfo function uses string field names (.Field100.) instead of the integer field names.

```
...
public void setupPDM() {//
  Create the definition with 3 fields and using string field names
  defn = new PDMDefinition(3, false);
  ...//
  Create information for a boolean, int32, and float field (all required)
  fldInfo100 = defn.addFieldInfo("Field100", PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo("Field101", PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo("Field102", PDMFieldType.FLOAT, true);
  ...
}
...
```

Retrieving FieldInfo from the Definition

At times, it may be easier to lookup the FieldInfo from the definition using the integer name (or string name if used). This eliminates the need to store the reference to the FieldInfo when getting or setting a field value in a message, but it does incur a performance penalty due to the lookup in the definition to retrieve the FieldInfo. Notice that there are no longer FieldInfo objects being used when calling addFieldInfo and a lookup is being done for each call to msg.getFieldValueAs∗ to retrieve the FieldInfo by integer name.

```
private PDMDefinition defn;
private PDMMessage msg;

public void setupPDM() {
  ...//
  Create information for a boolean, int32, and float field (all required)
  defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  defn.addFieldInfo(101, PDMFieldType.INT32, true);
  defn.addFieldInfo(102, PDMFieldType.FLOAT, true);
  ...
}

public void receiverUsePDM(byte[] buffer) {
  ...//
  Get each field value from the message
  fld100Val = msg.getFieldValueAsBoolean(defn.getFieldInfo(100));
  fld101Val = msg.getFieldValueAsInt32(defn.getFieldInfo(101));
  fld102Val = msg.getFieldValueAsFloat(defn.getFieldInfo(102));
}
```

Required and Optional Fields

When adding field information to a definition, you can indicate that the field is optional and may not be set for every message that uses the definition. Do this by passing false as the third parameter to the addFieldInfo function. Using required fields (fixed-required fields specifically) produces the best performance when serializing and deserializing messages, but causes an exception if all required fields are not set before serializing the message. Optional fields allow the concept of sending "null" as a value for a field by simply not setting that field value on the source side before serializing the message. However, after parsing a message, a receiver should check the isFieldValueSet function for an optional field before attempting to read the value from the field to avoid the exception mentioned above.

```
...
private PDMFieldInfo fldInfo103;
...
public void setupPDM() {
  ...//
  Create information for a boolean, int32, and float field (all required)//
   as well as an optional int8 field
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
```

```
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);
  fldInfo103 = defn.addFieldInfo(103, PDMFieldType.INT8, false);
  ...
}

public void sourceUsePDM() {
  ...//
  Set each field value in the message//
   except do not set the optional field
  msg.setFieldValue(fldInfo100, fld100Val);
  msg.setFieldValue(fldInfo101, fld101Val);
  msg.setFieldValue(fldInfo102, fld102Val);
  ...
}

...
private PDMFieldInfo fldInfo103;
...
public void setupPDM() {
  ...//
  Create information for a boolean, int32, and float field (all required)//
   as well as an optional int8 field
  fldInfo103 = defn.addFieldInfo(103, PDMFieldType.INT8, false);
  ...
}

public void receiverUsePDM(byte[] buffer) {
  ...
  byte fld103Val;
  ...
  if(msg.isFieldValueSet(fldInfo103)) {
    fld103Val = msg.getFieldValueAsInt8(fldInfo103);
  }
}
```

Fixed String and Fixed Unicode Field Types

A variable length string typically does not have the performance optimizations of fixed-required fields. However, by indicating "required", as well as the field type FIX_STRING or FIX_UNICODE and specifying an integer number of fixed characters, PDM sets aside an appropriate fixed amount of space in the message for that field and treats it as an optimized fixed-required field. Strings of a smaller length can still be set as the value for the field, but the message allocates the specified fixed number of bytes for the string. Specify Unicode strings in the same manner (with FIX_UNICODE as the type) and in "UTF-8" format.

```
...
private PDMFieldInfo fldInfo104;
...
public void setupPDM() {
  ...
  fldInfo104 = defn.addFieldInfo(104, PDMFieldType.FIX_STRING, 12, true);
  ...
}

public void sourceUsePDM() {
  ...
  String fld104Val = "Hello World!";//

  Set each field value in the message//
   except do not set the optional field
  msg.setFieldValue(fldInfo100, fld100Val);
  msg.setFieldValue(fldInfo101, fld101Val);
  msg.setFieldValue(fldInfo102, fld102Val);
  msg.setFieldValue(fldInfo104, fld104Val);
```

```
  ...
}

...
private PDMFieldInfo fldInfo104;
...
public void setupPDM() {
  ...
  fldInfo104 = defn.addFieldInfo(104, PDMFieldType.FIX_STRING, 12, true);
  ...
}
public void receiverUsePDM(byte[] buffer) {
  ...
  String fld104Val;
  ...

  fld104Val = msg.getFieldValueAsString(fldInfo104);
}
```

Variable Field Types

The field types of STRING, UNICODE, BLOB, and MESSAGE are all variable length field types. They do not require a length to be specified when adding field info to the definition. You can use a BLOB field to store an arbitrary binary objects (in Java as an array of bytes) and a MESSAGE field to store a PDMMessage object,

which enables "nesting" PDMMessages inside other PDMMessages. Creating and using a variable length string field is nearly identical to the previous fixed string example.

```
...
private PDMFieldInfo fldInfo105;
...
public void setupPDM() {
  ...
  fldInfo105 = defn.addFieldInfo(105, PDMFieldType.STRING, true);
  ...
}

public void sourceUsePDM() {
  ...
  String fld105Val = "variable length value";
  ...
  msg.setFieldValue(fldInfo105, fld105Val);
  ...
}

...
private PDMFieldInfo fldInfo105;
...
public void setupPDM() {
  ...
  fldInfo105 = defn.addFieldInfo(105, PDMFieldType.STRING, true);
  ...
}
public void receiverUsePDM(byte[] buffer) {
  ...
  String fld105Val;
  ...

  fld105Val = msg.getFieldValueAsString(fldInfo105);
}
```

Retrieve the BLOB field values with the getFieldValueAsBlob function, and the MESSAGE field values with the getFieldValueAsMessage function.

Array Field Types

For each of the scalar field types (fixed and variable length), a corresponding array field type uses the convention ∗_ARR for the type name (ex: BOOLEAN_ARR, INT32_ARR, STRING_ARR, etc.). This lets you set and get Java values such as an int[] or string[] directly into a single field. In addition, all of the array field types can specify a fixed number of elements for the size of the array when they are defined, or if not specified, behave as variable size arrays. Do this by passing an extra parameter to the addFieldInfo function of the definition.

To be treated as a fixed-required field, an array type field must be required as well as be specified as a fixed size array of fixed length elements. For instance, a required BOOLEAN_ARR field defined with a size of 3 would be treated as a fixed-required field. Also, a required FIX_STRING_ARR field defined with a size of 5 and fixed string length of 7 would be treated as a fixed-required field. However, neither a STRING_ARR field nor a BLOB_ARR field are treated as a fixed length field even if the size of the array is specified, since each element of the array can be variable in length. In the example below, field 106 and field 108 are both treated as fixed-required fields, but field 107 is not because it is a variable size array field type.

```
...
private PDMFieldInfo fldInfo106;
private PDMFieldInfo fldInfo107;
private PDMFieldInfo fldInfo108;
...

public void setupPDM() {
  ...//
  Create information for a boolean, int32, and float field (all required)//
   as well as an optional int8 field
  ...//
  A required, fixed size array of 3 boolean elements
  fldInfo106 = defn.addFieldInfo(106, PDMFieldType.BOOLEAN_ARR, true, 3);//
  An optional, variable size array of int32 elements
  fldInfo107 = defn.addFieldInfo(107, PDMFieldType.INT32_ARR, false);//
  A required, fixed size array of 2 element which are each 5 character strings
  fldInfo108 = defn.addFieldInfo(108, PDMFieldType.FIX_STRING_ARR, 5, true, 2);
  ...
}

public void sourceUsePDM() {
  ...//
  Example values for the message
  ...
  boolean fld106Val[] = {true, false, true};
  int fld107Val[] = {1, 2, 3, 4, 5};
  String fld108Val[] = {"aaaaa", "bbbbb"};//

  Set each field value in the message
  ...
  msg.setFieldValue(fldInfo106, fld106Val);
  msg.setFieldValue(fldInfo107, fld107Val);
  msg.setFieldValue(fldInfo108, fld108Val);
  ...
}

...
private PDMFieldInfo fldInfo106;
private PDMFieldInfo fldInfo107;
private PDMFieldInfo fldInfo108;
...
public void setupPDM() {
  ...//
  Create information for a boolean, int32, and float field (all required)//
   as well as an optional int8 field
  ...//
  A required, fixed size array of 3 boolean elements
  fldInfo106 = defn.addFieldInfo(106, PDMFieldType.BOOLEAN_ARR, true, 3);//
  An optional, variable size array of int32 elements
  fldInfo107 = defn.addFieldInfo(107, PDMFieldType.INT32_ARR, false);//
```

```
  A required, fixed size array of 2 element which are each 5 character strings
  fldInfo108 = defn.addFieldInfo(108, PDMFieldType.FIX_STRING_ARR, 5, true, 2);
  ...
}

public void receiverUsePDM(byte[] buffer) {
  ...//
  Values to be retrieved from the message
  ...
  boolean fld106Val[];
  int fld107Val[];
  String fld108Val[];//

  Deserialize the bytes into a message
  msg.parse(buffer);//

  Get each field value from the message
  ...
  fld106Val = msg.getFieldValueAsBooleanArray(fldInfo106);
  if(msg.isFieldValueSet(fldInfo107)) {
    fld107Val = msg.getFieldValueAsInt32Array(fldInfo107);
  }

  fld108Val = msg.getFieldValueAsStringArray(fldInfo108);
}
```

Definition Included In Message

Optionally, a PDM message can also include the definition when it is serialized to bytes. This enables receivers to parse a PDM message without having pre-defined knowledge of the message, although including the definition with the message affects message size and performance of message deserialization. Notice that the setIncludeDefinition function is called with an argument of true for a source that serializes the definition as part of the message.

```
private PDMDefinition defn;
private PDMMessage msg;

public void setupPDM() {//
  Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);
  ...//

  Finalize the definition and create the message
  defn.finalizeDef();
  msg = new PDMMessage(defn);//

  Set the flag to indicate that the definition should also be serialized
  msg.setIncludeDefinition(true);
}
...
```

For a receiver, the setupPDM function does not need to set any flags for the message but rather should define a message without a definition, since we assume the source provides the definition. If a definition is set for a message, it will attempt to use that definition instead of the definition on the incoming message (unless the ids are different).

```
private PDMDefinition defn;
private PDMMessage msg;

public void setupPDM() {//
  Don't define a definition//

  Create a message without a definition since the incoming message will have it
  msg = new PDMMessage();
}
...
```

The PDM Field Iterator

You can use the PDM Field Iterator to check all defined message fields to see if set, or to extract their values. You can extract a field value as an Object using this method, but due to the casting involved, we recommend you use the type specific get method to extract the exact value. Notice the use of field.isValueSet to check to see if the field value is set and the type specific get methods such as getBooleanValue and getFloatValue.

```
...

public void setupPDM() {//
  Create the definition with 3 fields and using int field names
  defn = new PDMDefinition(3, true);//

  Set the definition id and version
  defn.setId(1001);
  defn.setMsgVersMajor((byte)1);
  defn.setMsgVersMinor((byte)0);//

  Create information for a boolean, int32, and float field (all required)//
   as well as an optional int8 field
  fldInfo100 = defn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  fldInfo101 = defn.addFieldInfo(101, PDMFieldType.INT32, true);
  fldInfo102 = defn.addFieldInfo(102, PDMFieldType.FLOAT, true);
  fldInfo103 = defn.addFieldInfo(103, PDMFieldType.INT8, false);
  fldInfo104 = defn.addFieldInfo(104, PDMFieldType.FIX_STRING, 12, true);
  fldInfo105 = defn.addFieldInfo(105, PDMFieldType.STRING, true);//
  A required, fixed size array of 3 boolean elements
  fldInfo106 = defn.addFieldInfo(106, PDMFieldType.BOOLEAN_ARR, true, 3);//
  An optional, variable size array of int32 elements
  fldInfo107 = defn.addFieldInfo(107, PDMFieldType.INT32_ARR, false);//
  A required, fixed size array of 2 element which are each 5 character strings
  fldInfo108 = defn.addFieldInfo(108, PDMFieldType.FIX_STRING_ARR, 5, true, 2);//

  Finalize the definition and create the message
  defn.finalizeDef();
  msg = new PDMMessage(defn);
}

public void receiveAndIterateMessage(byte[] buffer) {
  msg.parse(buffer);
  PDMFieldIterator iterator = msg.createFieldIterator();
  PDMField field = null;
  while(iterator.hasNext()) {
    field = iterator.next();
    System.out.println("Field set? " +field.isValueSet());
    switch(field.getIntName()) {
      case 100:
        boolean val100 = field.getBooleanValue();
        System.out.println("Field 100's value is: " + val100);
        break;
      case 101:
        int val101 = field.getInt32Value();
        System.out.println("Field 101's value is: " + val101);
        break;
      case 102:
        float val102 = field.getFloatValue();
        System.out.println("Field 102's value is: " + val102);
        break;
      default://
        Casting to object is possible but not recommended
        Object value = field.getValue();
        int name = field.getIntName();
        System.out.println("Field " + name + "'s value is: " + value);
        break;
```

```
    }
  }
}
```

Sample Output (106, 107, 108 are array objects as expected):

```
Field set? true
Field 100's value is: true
Field set? true
Field 101's value is: 7
Field set? true
Field 102's value is: 3.14
Field set? false
Field 103's value is: null
Field set? true
Field 104's value is: Hello World!
Field set? true
Field 105's value is: Variable
Field set? true
Field 106's value is: [Z@527736bd
Field set? true
Field 107's value is: [I@10aadc97
Field set? true
Field 108's value is: [Ljava.lang.String;@4178460d
```

Using the Definition Cache

The PDM Definition Cache assists with storing and looking up definitions by their id and version. In some scenarios, it may not be desirable to maintain the references to the message and the definition from a setup phase by the application. A source could optionally create the definition during the setup phase and store it in the definition cache. At a later point in time, it could retrieve the definition from the cache and use it to create the message without needing to maintain any references to the objects.

```
public void createAndStoreDefinition() {
  PDMDefinition myDefn = new PDMDefinition(3, true);//
  Set the definition id and version
  myDefn.setId(2001);
  myDefn.setMsgVersMajor((byte)1);
  myDefn.setMsgVersMinor((byte)0);//

  Create information for a boolean, int32, and float field (all required)
  myDefn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  myDefn.addFieldInfo(101, PDMFieldType.INT32, true);
  myDefn.addFieldInfo(102, PDMFieldType.FLOAT, true);

  myDefn.finalizeDef();

  PDMDefinitionCache.getInstance().put(myDefn);
}

public void createMessageUsingCache() {
  PDMDefinition myFoundDefn = PDMDefinitionCache.getInstance().get(2001, 1, 0);
  if(myFoundDefn != null) {
    PDMMessage myMsg = new PDMMessage(myFoundDefn);//
    Get FieldInfo from defn and then set field values in myMsg//
    ...
  }
}
```

A more advanced use of the PDM Definition Cache is by a receiver which may need to receive messages with different definitions and the definitions are not being included with the messages. The receiver can create the definitions in advance and then set a flag that allows automatic lookup into the definition cache when parsing a message (which is not on by default). Before receiving messages, the receiver should do something similar to createAndStoreDefinition (shown below) to set up definitions and put them in the definition cache. Then the flag to allow automatic lookup should be set as shown below in the call to setTryToLoadDefFromCache(true). This allows the PDMMessage to be created without a definition and still successfully parse a message by leveraging the definition cache.

```
public void createAndStoreDefinition() {
  PDMDefinition myDefn = new PDMDefinition(3, true);//
  Set the definition id and version
  myDefn.setId(2001);
  myDefn.setMsgVersMajor((byte)1);
  myDefn.setMsgVersMinor((byte)0);//

  Create information for a boolean, int32, and float field (all required)
  myDefn.addFieldInfo(100, PDMFieldType.BOOLEAN, true);
  myDefn.addFieldInfo(101, PDMFieldType.INT32, true);
  myDefn.addFieldInfo(102, PDMFieldType.FLOAT, true);

  myDefn.finalizeDef();
  PDMDefinitionCache.getInstance().put(myDefn);//

  Create and store other definitions//
  ...
}

public void receiveKnownMessages(byte[] buffer) {
  PDMMessage myMsg = new PDMMessage();//
  Set the flag that enables messages to try//
   looking up the definition in the cache automatically//
   when parsing a byte buffer
  myMsg.setTryToLoadDefFromCache(true);
  myMsg.parse(buffer);

  if (myMsg.getDefinition().getId() == 2001
      && myMsg.getDefinition().getMsgVersMajor() == 1
      && myMsg.getDefinition().getMsgVersMinor() == 0) {
    PDMDefinition myDefn = PDMDefinitionCache.getInstance().get(2001, 1, 0);
    PDMFieldInfo fldInfo100 = myDefn.getFieldInfo(100);
    PDMFieldInfo fldInfo101 = myDefn.getFieldInfo(101);
    PDMFieldInfo fldInfo102 = myDefn.getFieldInfo(102);

    boolean fld100Val;
    int fld101Val;
    float fld102Val;//

    Get each field value from the message
    fld100Val = myMsg.getFieldValueAsBoolean(fldInfo100);
    fld101Val = myMsg.getFieldValueAsInt32(fldInfo101);
    fld102Val = myMsg.getFieldValueAsFloat(fldInfo102);

    System.out.println(fld100Val + " " + fld101Val + " " + fld102Val);
  }
}
```

7.10.4 Migrating from SDM

Applications using SDM with a known set of message fields are good candidates for migrating from SDM to PDM. With SDM, the source typically adds fields to an SDM message without a definition. But, as shown above in the PDM examples, creating/adding a PDM definition before adding field values is fairly straightforward.

However, certain applications may be incapable of building a definition in advance due to the ad-hoc nature of their messaging needs, in which case a self-describing format like SDM may be preferred.

Simple Migration Example

The following source code shows a basic application that serializes and deserializes three fields using SDM and PDM. The setup method in both cases initializes the object instances so they can be reused by the source and receiver methods.

The goal of the sourceCreateMessageWith functions is to produce a byte array by setting field values in a message object. With SDM, actual Field classes are created, values are set, the Field classes are added to a

Fields class, and then the Fields class is added to the SDMessage. With PDM, FieldInfo objects are created during the setup phase and then used to set specific values in the PDMMessage.

The goal of the receiverParseMessageWith functions is to produce a message object by parsing the byte array and then extract the field values from the message. With SDM, the specific field is located and casted to the correct field class before getting the field value. With PDM, the appropriate getFieldValueAs function is called with the corresponding FieldInfo object created during the setup phase to extract the field value.

```
public class Migration {//
  SDM Variables
  private LBMSDMessage srcSDMMsg;
  private LBMSDMessage rcvSDMMsg;//

  PDM Variables
  private PDMDefinition defn;
  private PDMFieldInfo fldInfo100;
  private PDMFieldInfo fldInfo101;
  private PDMFieldInfo fldInfo102;
  private PDMMessage srcPDMMsg;
  private PDMMessage rcvPDMMsg;

  public static void main(String[] args) {
    Migration app = new Migration();
    System.out.println("Setting up PDM Definition and Message");
    app.setupPDM();
    System.out.println("Setting up SDM Messages");
    app.setupSDM();

    byte[] sdmBuffer;
    sdmBuffer = app.sourceCreateMessageWithSDM();
    app.receiverParseMessageWithSDM(sdmBuffer);

    byte[] pdmBuffer;
    pdmBuffer = app.sourceCreateMessageWithPDM();
    app.receiverParseMessageWithPDM(pdmBuffer);
  }

  public void setupSDM() {
    rcvSDMMsg = new LBMSDMessage();
    srcSDMMsg = new LBMSDMessage();
  }

  public void setupPDM() {//
    Create the definition with 3 fields and using int field names
    defn = new PDMDefinition(3, false);//

    Set the definition id and version
    defn.setId(1001);
    defn.setMsgVersMajor((byte)1);
    defn.setMsgVersMinor((byte)0);//

    Create information for a boolean, int32, and float field (all required)//
     as well as an optional int8 field
    fldInfo100 = defn.addFieldInfo("Field100", PDMFieldType.INT8, true);
    fldInfo101 = defn.addFieldInfo("Field101", PDMFieldType.INT16, true);
    fldInfo102 = defn.addFieldInfo("Field102", PDMFieldType.INT32, true);//
```

```
  Finalize the definition and create the message defn.finalizeDef();
  srcPDMMsg = new PDMMessage(defn);
  rcvPDMMsg = new PDMMessage(defn);
}

public byte[] sourceCreateMessageWithSDM() {
  byte[] buffer = null;

  LBMSDMField fld100 = new LBMSDMFieldInt8("Field100", (byte)0x42);
  LBMSDMField fld101 = new LBMSDMFieldInt16("Field101", (short)0x1ead);
  LBMSDMField fld102 = new LBMSDMFieldInt32("Field102", 12345);
  LBMSDMFields fset = new LBMSDMFields();

  try {
    fset.add(fld100);
    fset.add(fld101);
    fset.add(fld102);
  } catch (LBMSDMException e) {
    System.out.println ( e );
  }

  srcSDMMsg.set(fset);
  try {
    buffer = srcSDMMsg.data();
  } catch (IndexOutOfBoundsException e) {
    System.out.println ( "SDM Exception occurred during build of message:" );
    System.out.println ( e.toString() );
  } catch (LBMSDMException e) {
    System.out.println ( e.toString() );
  }
  return buffer;
}

public byte[] sourceCreateMessageWithPDM() {//
  Set each field value in the message
  srcPDMMsg.setFieldValue(fldInfo100, (byte)0x42);
  srcPDMMsg.setFieldValue(fldInfo101, (short)0x1ead);
  srcPDMMsg.setFieldValue(fldInfo102, 12345);//

  Serialize the message to bytes
  byte[] buffer = srcPDMMsg.toBytes();
  return buffer;
}

public void receiverParseMessageWithSDM(byte[] buffer) {//
  Values to be retrieved from the message byte fld100Val;
  short fld101Val;
  int fld102Val;//

  Deserialize the bytes into a message
  try {
    rcvSDMMsg.parse(buffer);
  } catch (LBMSDMException e) {
    System.out.println(e.toString());
  }

  LBMSDMField fld100 = rcvSDMMsg.locate("Field100");
  LBMSDMField fld101 = rcvSDMMsg.locate("Field101");
  LBMSDMField fld102 = rcvSDMMsg.locate("Field102");//

  Get each field value from the message
  fld100Val = ((LBMSDMFieldInt8)fld100).get();
```

```
    fld101Val = ((LBMSDMFieldInt16)fld101).get();
    fld102Val = ((LBMSDMFieldInt32)fld102).get();

    System.out.println("SDM Results: Field100=" + fld100Val +
                       ", Field101=" + fld101Val +
                       ", Field102=" + fld102Val);
  }

  public void receiverParseMessageWithPDM(byte[] buffer) {//
    Values to be retrieved from the message
    byte fld100Val;
    short fld101Val;
    int fld102Val;//

    Deserialize the bytes into a message
    rcvPDMMsg.parse(buffer);//

    Get each field value from the message
    fld100Val = rcvPDMMsg.getFieldValueAsInt8(fldInfo100);
    fld101Val = rcvPDMMsg.getFieldValueAsInt16(fldInfo101);
    fld102Val = rcvPDMMsg.getFieldValueAsInt32(fldInfo102);

    System.out.println("PDM Results: Field100=" + fld100Val +
                       ", Field101=" + fld101Val +
                       ", Field102=" + fld102Val);
  }
}
```

Notice that with sourceCreateMessageWithSDM function, the three fields (name and value) are created and added to the fset variable, which is then added to the SDM message. On the other hand, the sourceCreateMessage↩ WithPDM function uses the FieldInfo object references to add the field values to the message for each of the three fields.

Also notice that the receiverParseMessageWithSDM requires a cast to the specific field class (like LBMSDMField↩ Int8) once the field has been located. After the cast, calling the get method returns the expected value. On the other hand the receiverParseMessageWithPDM uses the FieldInfo object reference to directly retrieve the field value using the appropriate getFieldValueAs∗ method.

SDM Raw Classes

Several SDM classes with Raw in their name could be used as the value when creating an LBMSDMField. For example, an LBMSDMRawBlob instance could be created from a byte array and then that the LBMSDMRawBlob could be used as the value to a LBMSDMFieldBlob as shown in the following example.

```
byte[] blob = new byte[25];
LBMSDMRawBlob rawSDMBlob = new LBMSDMRawBlob(blob);
try {
  LBMSDMField fld103 = new LBMSDMFieldBlob("Field103",rawSDMBlob);
} catch (LBMSDMException e1) {
  System.out.println(e1);
}
```

The actual field named "Field103" is created in the try block using the rawSDMBlob variable which has been created to wrap the blob byte array. This field can be added to a LBMSDMFields object, which then uses it in a LBMSD↩ Message.

In PDM, there are no "Raw" classes that can be created. When setting the value for a field for a message, the appropriate variable type should be passed in as the value. For example, setting the field value for a BLOB field would mean simply passing the byte array directly in the setValue method as shown in the following code snippet since the field is defined as type BLOB.

```
private PDMFieldInfo fldInfo103;
public void setupPDM() {
  ...
  fldInfo103 = defn.addFieldInfo("Field103", PDMFieldType.BLOB, true);
```

```
  ...
  byte[] blob = new byte[25];

  srcPDMMsg.setFieldValue(fldInfo103, blob);
  ...
}
```

The PDM types of DECIMAL, TIMESTAMP, and MESSAGE expect a corresponding instance of PDMDecimal, 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);
```

7.11 Sending to Sources

There are many use cases where a subscriber application wants to send a message to a publisher application. For example, a client application which subscribes to market data may want to send a refresh request to the publishing feed handler. While this is possible to do with normal sources and receivers, UM supports a streamlined method of doing this.

As of UM version 6.10, a Source String can be used as a destination for sending a unicast immediate message. The UM library will establish a TCP connection to the publisher's context via its *request port*. The publishing application can receive this message either from a normal Receiver Object, or from a context immediate message callback via configuration options **immediate_message_topic_receiver_function (context)** or **immediate_message_receiver_function (context)** (for topicless messages).

7.11.1 Source String from Receive Event

A receiving application's receiver callback function can obtain a source's source string from the message structure. However, that string is not suitable to being passed directly to the unicast immediate message send function.

Here's a code fragment in C for receiving a message from a source, and sending a message back to the originating source. For clarity, error detection and handling code is omitted.

```
int user_receiver_callback(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)
{
  ...
  switch (msg->type) {
    ...
  case LBM_MSG_DATA:/
    * user code which processes received message and sets up "msg_for_src" */
    .../
    * A valid UIM destination is "SOURCE:" + source string. */
    char destination[LBM_MSG_MAX_SOURCE_LEN + 8];
    strcpy(destination, "SOURCE:");
    strcat(destination, msg->source);

    err = lbm_unicast_immediate_message(ctx, destination, NULL, /  * no topic */
                                        msg_for_src, sizeof(msg_for_src),
                                        LBM_SRC_NONBLOCK); / * Called from context
                                            thread. */
    ...
  } / * switch msg->type */
```

```
    ...
} / * user_receiver_callback */
```

The **lbm_msg_t** structure supplies the source string, and **lbm_unicast_immediate_message()** is used to send a topicless immediate message to the source's context. Alternatively, a request message could be sent with **lbm_unicast_immediate_request()**. If the receive events are delivered without an event queue, then **LBM_SRC_NONBLOCK** is needed.

The example above uses the **LBM_MSG_DATA** message type. Most receiver event (message) types also contain a valid source string. Other likely candidates for this use case might be: **LBM_MSG_BOS**, **LBM_MSG_UNRECOVERABLE_LOSS**, **LBM_MSG_UNRECOVERABLE_LOSS_BURST**.

Note that in this example, a topicless message is sent. This requires the publishing application to use the **immediate_message_receiver_function (context)** option to set up a callback for receipt of topicless immediate messages. Alternatively, a topic name can be supplied to the unicast immediate message function, in which case the publishing application would either create a normal Receiver Object for that topic, or would configure a callback with **immediate_message_topic_receiver_function (context)**.

A Java program obtains the source string via **com::latencybusters::lbm::LBMMessage::source**, and sends topicless unicast immediate messages via **com::latencybusters::lbm::LBMContext::sendTopicless**.

A .NET implementation is essentially the same as Java.

7.11.2 Source String from Source Notification Function

Some subscribing applications need to send a message to the publisher as soon as possible after the publisher is subscribed. Receiver events can sometimes take significant time to be delivered. The source string can be obtained via the **source_notification_function (receiver)** configuration option. This defines a callback function which is called at the start of the process of subscribing to a source.

Here's a code fragment in C for sending a message to a newly-discovered source. For clarity, error detection and handling code is omitted.

During initialization, when the receiver is defined, the callback must be configured using the **lbm_rcv_src_notification_func_t_stct** structure:

```
lbm_rcv_src_notification_func_t src_notif_callback_info;
src_notif_callback_info.create_func = src_notif_callback_create; / * User
    function. */
src_notif_callback_info.delete_func = src_notif_callback_delete; / * User
    function. */
src_notif_callback_info.clientd = NULL; /  * Can be user's receiver-specific
    state. */
...
lbm_rcv_topic_attr_t *rcv_topic_attr;
err = lbm_rcv_topic_attr_create(&rcv_topic_attr);

err = lbm_rcv_topic_attr_setopt(rcv_topic_attr, "source_notification_function",
                                &src_notif_callback_info,
                                    sizeof(src_notif_callback_info));

lbm_topic_t *receiver_topic;
err = lbm_rcv_topic_lookup(&receiver_topic, ctx, receiver_topic_name,
    rcv_topic_attr);

lbm_rcv_t *receiver;
err = lbm_rcv_create(&receiver, ctx, receiver_topic, ...);
```

This creates the Receiver Object with the source notification callback configured. Note that the source notification callback has both a create and a delete function, to facilitate state management by the user.

```
void * src_notif_callback_create(const char *source_name, void *clientd)
{/
  * This function is called when the subscription is being set up. *//

  * user code which sets up "msg_for_src" */
  .../
  * A valid UIM destination is "SOURCE:" + source string. */
  char destination[LBM_MSG_MAX_SOURCE_LEN + 8];
  strcpy(destination, "SOURCE:");
  strcat(destination, source_name);

  err = lbm_unicast_immediate_message(ctx, destination, NULL, /  * no topic */
                                     msg_for_src, sizeof(msg_for_src),
                                     LBM_SRC_NONBLOCK); / * Called from context
                                         thread. */
  ...
  return NULL; / * Can be per-source state. */
} / * src_notif_callback_create */


int src_notif_callback_delete(const char *source_name, void *clientd, void
    *source_clientd) {/
  * This function not used for anything in this example, but could be used to
   * to clean up per-source state. */
  return 0;
} / * src_notif_callback_delete */
```

A Java program configures the source notification callback via **com::latencybusters::lbm::LBMReceiverAttributes::setSourceNotificationCallbacks**.

A .NET implementation is essentially the same as Java.

7.11.3 Sending to Source Readiness

In most use cases for sending messages to a source, there is an implicit assumption that a subscribing receiver is fully set up and ready to receive messages from the publisher. However, due to the asynchronous nature of UM, there is no straight-forward way for a receiver to know the earliest point in time when messages sent by the source will be delivered to the receiver. For example, in a routed network (using the UM Router), a receiver might deliver BOS to the application, but that just means that the connection to the proper UM Router is complete. There could still be delays in the entire end-to-end path being able to deliver messages.

Also, be aware that although unicast immediate messages are delivered via TCP, these messages are not guaranteed. Especially in a routed network, there exists the possibility that a message will fail to reach the publisher.

In most cases, the immediate message is received by the publisher, and by the time the publisher reacts, the end-to-end source-to-receiver path is active. However, in the unlikely event that something goes wrong, a subscribing application should implement a timeout/retry mechanism. This advice is not specific to the "sending to source" use cases, and should be built into any kind of request/response-oriented use case.

7.12 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:

- Configuring transport information into sending and receiving applications.
- 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** for more information

Note

With the UMQ product, you cannot use MIM with Queuing.

7.12.1 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 (context)** which defaults to 30 seconds. A subsequent send creates a new Transport Session. Due to the possibility of head-loss in the switch, it is recommended that 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.

7.12.2 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.

7.12.3 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.

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.

7.12.4 MIM and Wildcard Receivers

When an application receives an immediate message, it's topic is hashed to see if there is at least one regular (non-wildcard) receiver object listening to the topic. If so, then MIM delivers the message data to the list of receivers.

However, if there are no regular receivers for that topic in the receive hash, MIM runs the message topic through all existing wildcard patterns and delivers matches to the appropriate wildcard receiver objects without creating sub-receivers. The next MIM message received for the same topic will again be run through all existing wildcard patterns. This can consume significant CPU resources since it is done on a per-message basis.

7.12.5 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.

7.12.6 MIM Configuration

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

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**

7.12.7 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**

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
```

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...
```

7.13 Spectrum

UM Spectrum, which refers to a "spectrum of channels", allows the application designer to sub-divide a topic into any number of channels, which can be individually subscribed to by a receiving application. This provides an extra level of message filtering.

The sending application first allocates the desired number of source channel objects using **lbm_src_channel_create()**. Then it creates a topic source in the normal way. Finally, the application sends messages using **lbm_src_send_ex()**, specifying the source channel object in the **lbm_src_send_ex_info_t**'s channel_info field.

A receiving application first creates a topic receiver in the normal way. Then it subscribes to channels using **lbm_rcv_subscribe_channel()** or lbm_wrcv_subscribe_channel(). Since each channel requires a different receiver callback, the receiver application can achieve more granular filtering of messages. Moreover, messages are received in-order across channels since all messages are part of the same topic stream.

It should be noted that a regular topic receiver (one for which no spectrum channels are subscribed) delivers all received messages from a matching spectrum topic source to the receiver's callback without creating the channel_info object.

You can accomplish the same level of filtering with a topic space design that creates separate topics for each channel, however, UM cannot guarantee the delivery of messages from multiple sources/topics in any particular order. Not only can UM Spectrum deliver the messages over many channels in the order they were sent by the source, but it also reduces topic resolution traffic since UM advertises only topics, not channels.

Note

With the UMQ product, you cannot use UM Spectrum with Queuing.

7.13.1 Spectrum Performance Advantages

The use of separate callbacks for different channels improves filtering and also relieves the source application of the task of including filtering information in the message data.

Java and .NET performance also receives a boost because messages not of interest can be discarded before they transition to the Java or .NET level.

7.13.2 Spectrum Configuration Options

Spectrum's default behavior delivers messages on any channels the receiver has subscribed to on the callbacks specified when subscribing, and all other messages on the receiver's default callback. This behavior can be changed with the following configuration options.

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

7.13.3 Smart Sources and Spectrum

Smart Sources support Spectrum, but via different API functions. You need to tell UM that you intend to use spectrum at Smart Source creation time using the **smart_src_enable_spectrum_channel (source)** configuration option. This pre-allocates space in the message header for the spectrum channel.

With Smart Sources, there is no need to allocate a Spectrum source object with **lbm_src_channel_create()**. Instead, you simply set the **LBM_SSRC_SEND_EX_FLAG_CHANNEL** flag and the spectrum channel number in the **lbm_ssrc_send_ex_info_t** passed to the **lbm_ssrc_send_ex()** API function. For example:

```
lbm_ssrc_send_ex_info_t ss_send_info;
memset((char *)&ss_send_info, 0, sizeof(ss_send_info));/
* If this flag had been cleared previously, must set it. */
ss_send_info.flags |= LBM_SSRC_SEND_EX_FLAG_CHANNEL;
ss_send_info.channel = desired_channel_number;

err = lbm_ssrc_send_ex(ss, msg_buff, msg_size, 0, &ss_send_info);
```

When a Smart Source is created with Spectrum enabled, it is possible to send messages without a Spectrum channel, either by clearing the LBM_SSRC_SEND_EX_FLAG_CHANNEL flag in **lbm_ssrc_send_ex_info_t**, or by simply not supplying a **lbm_ssrc_send_ex_info_t** object by passing NULL for the `info` parameter. This suppresses all features enabled by that structure.

Note

> If using both Spectrum and Message Properties with a single Smart Source, there is an added restriction: it is not possible to send a message omitting only one of those features. I.e. if both are enabled when the Smart Source is created, it is not possible to send a message with a message property and not a channel, and it is not possible to send a message with a channel and not a property. This is because the message header is defined at Smart Source creation, and the header either must contain both or neither.

7.14 Hot Failover (HF)

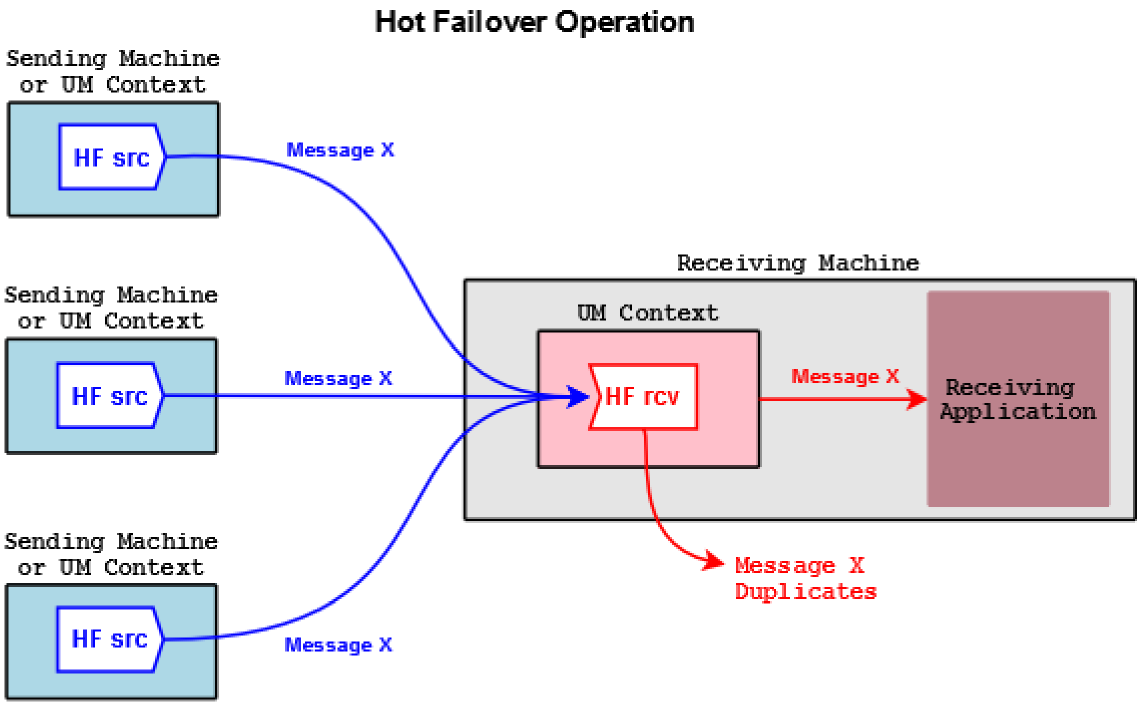
UM Hot Failover (HF) lets you implement sender redundancy in your applications. You can create multiple HF senders in different UM contexts, or, for even greater resiliency, on separate machines. There is no hard limit to the number of HF sources, and different HF sources can use different transport types.

Note

> With the UMQ product, you cannot use Hot Failover with Queuing.

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.

7.14.1 Implementing Hot Failover Sources

You create Hot Failover sources with **lbm_hf_src_create()**. This returns a source object with internal state information that lets it send HF messages. You delete HF sources with the **lbm_src_delete()** function.

HF sources send HF messages via **lbm_hf_src_send_ex()** or **lbm_hf_src_sendv_ex()**. These functions take a sequence number, supplied via the exinfo object, that HF receivers use to identify the same message sent from different HF sources. The exinfo has an hf_sequence_number, with a flag (LBM_SRC_SEND_EX_FLAG_HF_32 or LBM_SRC_SEND_EX_FLAG_HF_64) that identifies whether it's a 32- or 64-bit number. Each HF source sends the same message content for a given sequence number, which must be coordinated by your application.

If the source needs to restart its sequence number to an earlier value (e.g. start of day; not needed for normal wraparound), delete and re-create the source and receiver objects. Without re-creating the objects, the receiver sees the smaller sequence number, assumes the data are duplicate, and discards it. In (and only in) cases where this cannot be done, use **lbm_hf_src_send_rcv_reset()**.

Note

> Your application must synchronize calling **lbm_hf_src_send_ex()** or **lbm_hf_src_sendv_ex()** with all threads sending on the same source. (One symptom of not doing so is messages appearing at the receiver as inside intentional gaps and being erroneously discarded.)

Please be aware that non-HF receivers created for an HF topic receive multiple copies of each message. We recommend you establish local conventions regarding the use of HF sources, such as including "HF" in the topic name.

For an example source application, see `lbmhfsrc.c`.

7.14.2 Implementing Hot Failover Receivers

You create HF receivers with **lbm_hf_rcv_create()**, and delete them using **lbm_hf_rcv_delete()** and **lbm_hf_rcv_delete_ex()**.

Incoming messages have an hf_sequence_number field containing the sequence number, and a message flag (LBM_MSG_FLAG_HF_32 or LBM_MSG_FLAG_HF_64) noting the bit size.

Note

> Previous UM versions used sequence_number for HF message identification. This field holds a 32-bit value and is still set for backwards compatibility, but if the HF sequence numbers are 64-bit lengths, this non-HF sequence number is set to 0. Also, you can retrieve the original (non-HF) topic sequence number via lbm_msg_retrieve_original_sequence_number() or, in Java and .NET, via LBMMessage.osqn().

For the maximum time period to recover lost messages, the HF receiver uses the minimum of the LBT-RM and LBT-RU NAK generation intervals (**transport_lbtrm_nak_generation_interval (receiver)**, **transport_lbtru_nak_generation_interval (receiver)**). Each transport protocol is configured as normal, but the lost message recovery timer is the minimum of the two settings.

Some lbm_msg_t objects coming from HF receivers may be flagged as having "passed through" the HF receiver. This means that the message has not been ordered with other HF messages. These messages have the LBM_MSG_FLAG_HF_PASS_THROUGH flag set. UM flags messages sent from HF sources using **lbm_src_send()** in this manner, as do all non-HF sources. Also, UM flags EOS, no source notification, and requests in this manner as well.

For an example receiver application, see `lbmhfrcv.c`.

7.14.3 Implementing Hot Failover Wildcard Receivers

To create an HF wildcard receiver, set option **hf_receiver (wildcard_receiver)** to 1, then create a wildcard receiver with **lbm_wildcard_rcv_create()**. This actually creates individual HF receivers on a per-topic basis, so that each topic can have its own set of HF sequence numbers. Once the HF wildcard receiver detects that all sources for a particular topic are gone it closes the individual topic HF receivers and discards the HF sequence information (unlike a standard HF receiver). You can extend or control the delete timeout period of individual HF receivers with option **resolver_no_source_linger_timeout (wildcard_receiver)**.

7.14.4 Java and .NET

For information on implement the HF feature in a Java application, go to UM Java API and see the documentation for classes **com::latencybusters::lbm::LBMHotFailoverReceiver** and **com::latencybusters::lbm::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.

7.14.5 Using Hot Failover with Persistence

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

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

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

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

7.14.6 Hot Failover Intentional Gap Support

UM supports intentional gaps in HF message streams. Your HF sources can supply message sequence numbers with number gaps up to 1073741824. HF receivers automatically detect the gaps and consider any missing message sequence numbers as not sent and do not attempt recovery for these missing sequence numbers. See the following example.

1. HF source 1 sends message sequence numbers: 10, 11, 12, 13, 25, 26, 38
2. HF source 2 sends message sequence numbers: 10, 11, 12, 13, 25, 26, 38

HF receiver 1 receives message sequence numbers in order with no pause between any messages: 10, 11, 12, 13, 25, 26, 38

7.14.7 Hot Failover Optional Messages

Hot Failover sources can send optional messages that HF receivers can be configured to receive or not receive (**hf_optional_messages (receiver)**). HF receivers detect an optional message by checking **lbm_msg_t.flags** for LBM_MSG_FLAG_HF_OPTIONAL. HF sources indicate an optional message by passing **LBM_SRC_SEND_EX_FLAG_HF_OPTIONAL** in the **lbm_src_send_ex_info_t.flags** field to **lbm_hf_src_send_ex()** or **lbm_hf_src_sendv_ex()**. In the examples below, optional messages appear with an "o" after the sequence number.

1. HF source 1 sends message sequence numbers: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o, 19o, 20
2. HF source 2 sends message sequence numbers: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o, 19o, 20

HF receiver 1 receives: 10, 11, 12, 13o, 14o, 15, 16o, 17o, 18o, 19o, 20

HF receiver 2, configured to ignore optional messages, receives: 10, 11, 12, 15, 20

7.14.8 Using Hot Failover with Ordered Delivery

An HF receiver takes some of its operating parameters directly from the receive topic attributes. The **ordered_delivery (receiver)** setting indicates the ordering for the HF receiver.

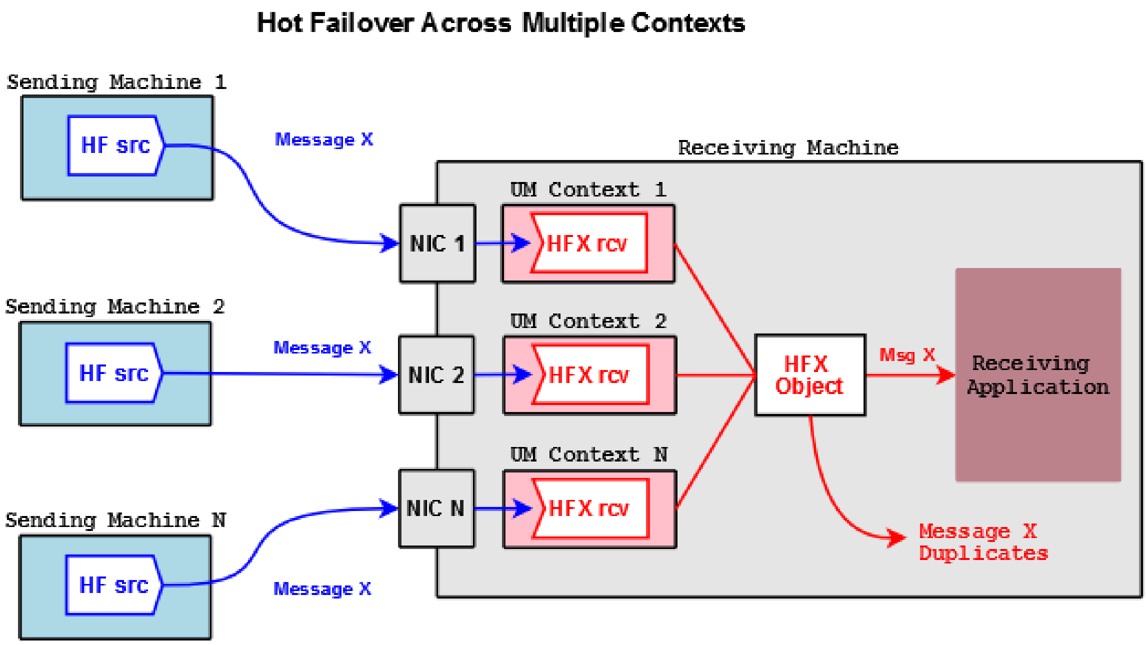
Note

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

7.14.9 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 **Hot Failover Operation Options**, especially HFX-specific options.

7.15 Daemon Statistics

The Persistence Store daemon and the UM Router daemon each have a simple web server which provides operational information. This information is important for monitoring the operation and performance of these daemons. However, while the web-based presentation is convenient for manual, on-demand monitoring, it is not suitable for automated collection and recording of operational information for historical analysis.

Starting with UM version 6.11, a feature called "Daemon Statistics" has been added to the Store and Router daemons. The Stateful Resolver Service (SRS), added in UM version 6.12, supports Daemon Statistics only (no web server). The Daemon Statistics feature supports the background publishing of their operational information via UM messages. Monitoring systems can now subscribe to this information in much the same way that UM transport statistics can be subscribed.

While the information published by the Store, UM Router, and SRS daemon statistics differ in their content, the general feature usage is the same between them. When the feature is configured, the daemon will periodically collect and publish its operational information.

The following sections give general information which is common across daemons, followed by links to daemon-specific details.

7.15.1 Daemon Statistics Structures

The operational information is published as messages of different types sent over a normal UM topic source (topic name configurable). For the Store and Router daemons, each message is in the form of a binary, C-style data structure. For the SRS service, the messages are formatted as JSON.

There are generally two categories of messages: *config* and *stats*. A given instance of a category "config" message does not have content which changes over time. An instance of a category "stats" message has content that *does* change over time. The daemon-specific documentation indicates which messages are in which category.

Each message type is configured for a publishing interval. However, config category messages are treated differently than stat. When the publishing interval for a given instance of a "config" message expires, the message is re-published unconditionally. These publishing intervals are typically set to long periods. TBD: is SRS different from Store/DRO? However, when the publishing interval for a "stats" message expires, the message is checked to see if its content has materially changed since the last interval. If not, then the message is *not* republished. The publishing interval for a stat message is typically set to shorter periods to see those changes as they occur.

Finally, note that while the contents of a given instance of a config message does not change over time, new instances of the message type can be sent as a result of state changes. For example, a new instance of **umestore_repo_dmon_config_msg_t** is published each time a new source registers with the store.

More detailed information is available in the daemon-specific documentation referenced below.

7.15.2 Daemon Statistics Binary Data

For the Store and Router daemons, the messages published are in binary form and map onto the C data structures defined for each message type.

For the SRS service, the messages are formatted as JSON, so this section does not apply to the SRS.

The byte order of the structure fields is defined as the host endian architecture of the publishing daemon. Thus, if a monitoring host receiving the messages has the same endian architecture, the binary structures can be used directly. If the monitoring host has the opposite endian architecture, the receiver must byte-swap the fields.

The message structure is designed to make it possible for a monitoring application to detect a mismatch in endian architecture. Detection and byte swapping is demonstrated with daemon-specific example monitoring applications.

More detailed information is available in the daemon-specific documentation referenced below.

7.15.3 Daemon Statistics Versioning

For the Store and Router daemons, each message sent by the daemon consists of a standard header followed by a message-type-specific set of fields. The standard header contains a `version` field which identifies the version of the C include file used to build the daemon.

For the SRS service, the messages are formatted as JSON, so this section does not apply to the SRS.

For example, the Store daemon is built with the include file **umedmonmsgs.h**. With each daemon statistics message sent by the Store daemon, it sets the header version field to **LBM_UMESTORE_DMON_VERSION**. With each new release of the UM package, if that include file changes in a substantive way, the value of **LBM_UMESTORE_DMON_VERSION** is increased. In this way, a monitoring application can determine if it is receiving messages from a store daemon whose data structures match the monitoring application's structure definitions.

More detailed information is available in the daemon-specific documentation referenced below.

7.15.4 Daemon Statistics Requests

The daemon can optionally be configured to accept command-and-control requests from monitoring applications. There are two categories of these requests: *"snapshot"* and *"config"*. "Snapshot" requests tell the daemon to immediately republish the desired stats and/or configs without waiting until the next publishing interval. These requests might be sent by a monitoring application which has only just started running and needs a full snapshot of the operational information. "Config" requests tell the daemon to modify an operational parameter of the running daemon.

The monitoring application sends a request to the daemon, and the daemon sends status messages in response. The exchanges are made via standard UM Request/Response messaging.

For the Store and Router daemons, The request message is formatted as a simple ASCII string. For the SRS service, the request message is formatted as a JSON message. The request is sent as a non-topic unicast immediate request message. The daemon reacts by parsing the request and sending a UM response with a success/failure response. If the request was parsed successfully, the daemon then performs the requested operation (republishing the data or modifying the operational parameter). There are daemon-specific example applications which demonstrate the use of this request feature.

More detailed information is available in the daemon-specific documentation referenced below.

7.15.5 Daemon Statistics Details

For details on the Persistent Store's daemon statistics feature, see **Store Daemon Statistics**.

For details on the UM Router's daemon statistics feature, see **UM Router Daemon Statistics**.

For details on the SRS's daemon statistics feature, see SRS Daemon Statistics.

Chapter 8

Advanced Optimizations

The internal design of UM has many compromises between performance and flexibility. For example, there are critical sections which maintain state information which must be kept internally consistent. Since UM allows the application the flexibility of multi-threaded use, those critical sections are protected with Mutex locks. These locks add very little overhead to UM's execution, but "very little" is not the same as "zero". The use of locks is a compromise between efficiency and flexibility. Similar lines of reasoning explain why UM makes use of dynamic memory (malloc and free), and bus-interlocked read/modify/write operations (e.g. atomic increment).

UM provides configuration options which improve efficiency, at the cost of reduced application design flexibility. Application designers who are able to constrain their programs within certain restrictions can take advantage of improved performance and reduced latency outliers.

- Receive Buffer Recycling - Receive-side latency reduction.
- Single Receiving Thread - Receive-side latency reduction.
- lbm_context_process_events_ex - Receive-side latency reduction.
- Receive Multiple Datagrams - Receive-side latency reduction.
- Smart Sources - Send-side latency reduction.
- Zero-Copy Send API - Alternative Send-side latency reduction. See also Comparison of Zero Copy and Smart Sources.

8.1 Receive Buffer Recycling

By default, the UM receive code base allocates a fresh buffer for each received datagram. This allows the user a great degree of threading and buffer utilization flexibility in the application design.

For transport types RM (reliable multicast), RU (Reliable Unicast), and IPC (shared memory), you can set a configuration option to enable reuse of receive buffers, which can avoid per-message dynamic memory calls (malloc/free). This produces a modest reduction in average latency, but more importantly, can significantly reduce occasional latency outliers.

See the configuration options:

- RM - **transport_lbtrm_recycle_receive_buffers (context)**
- RU - **transport_lbtru_recycle_receive_buffers (context)**
- IPC - **transport_lbtipc_recycle_receive_buffers (context)**

Note that setting the option does not guarantee the elimination of per-message malloc and free except in fairly restrictive use cases.

8.1.1 Receive Buffer Recycling Restrictions

There are no hard restrictions to enabling buffer recycling. I.e. it is not functionally not compatible with any use patterns or UM features. However, some use patterns will prevent the recycling of the receive buffer, and therefore not deliver the benefit, even if the configuration option is set.

- **Event Queues** - Event Queues prevent the recycling of receive buffers. When the UM library transfers a received message to an event queue for later processing, it allocates (malloc) a new message receive buffer.
- **Message Object Retention** - Message retention prevents the recycling. For context-thread receive message callbacks, the act of retaining a message allocates (mallocs) a new message receive buffer.
- **Persistence** - For a persistent receiver, enabling receive buffer recycling will *reduce* dynamic memory usage (malloc/free), but does not eliminate it. Certain persistence-related features require the use of dynamic memory.
- **Packet Loss** - Applications typically use **Ordered Delivery**. When packets are lost, UM needs to internally retain newly received messages so that they can be delivered after the missing messages are retransmitted. This internal retention prevents the newly received message buffers from being recycled.
- **Message Fragmentation and Reassembly** - Large application messages must be split into smaller fragments and sent serially. The receiver must internally retain these fragments so that the original large message can be reassembled and delivered to the application. This internal retention prevents the fragment message buffers from being recycled.

Note that in spite of the restrictions that can prevent recycling of receive message buffers, UM dynamically takes advantage of recycling as much as it can. E.g. if there is a loss event which suppresses recycling while waiting for retransmission, once the gap is filled and pending messages are delivered, UM will once again be able to recycle its receive buffers.

Of specific interest for *persistent* receivers is the use of **Explicit Acknowledgments**, either to batch ACKs, or simply defer them. Instead of retaining the messages, which prevents message buffer recycling, you can extract the ACK information from a message and allow the return from the receiver callback to delete and recycle the message buffer without acknowledging it.

See **Object-free Explicit Acknowledgments** for details.

8.2 Single Receiving Thread

This feature optimizes the execution of UM receive-path code by converting certain thread-safe operations to more-efficient thread-unsafe operations. For example, certain bus-locked operations (e.g. atomic increment) are replaced by non-bus-locked equivalents (e.g. non-atomic increment). This can reduce the latency of delivering received messages to the application, but does so at the expense of thread safety.

This feature is often used in conjunction with the Context Lock Reduction feature.

The **transport_session_single_receiving_thread (context)** configuration option enables this feature.

Except as listed in the restrictions below, the Single Receiving Thread feature should be compatible with all other receive-side UM features.

8.2.1 Single Receiving Thread Restrictions

It is very important for applications using this feature to be designed within certain restrictions.

- **Threading** - The intended use case is for each received message to be fully processed by the UM thread that delivers the message to the application. Note that the Transport Services Provider (XSP) feature *is* compatible with the Single Receiving Thread feature.
- **No Event Queues** - Event queues cannot be used with Single Receiving Thread.
- **Message Object Retention** - Most traditional uses of message retention are related to giving a message to an alternate thread for processing. This is not compatible with Single Receiving Thread feature.

  However, there are some use cases where message retention is viable when used with Single Receiving Thread: when a message must be held for *future* processing, and that processing will be done by the same thread.

  For example, a persistent application might use **Explicit Acknowledgments** to delay message acknowledgement until the application completes a handshake with a remote service. As long as it is the same thread which initially receives and retains the message as that which completes the explicit acknowledgement of the message, it is supported to use message retain / message delete.

  Note

  > If the Transport Services Provider (XSP) feature is used, care must be taken to ensure that the same XSP thread is used to perform all processing for a received message. I.e. a different XSP or the main context may not be used to complete processing on a deferred retained message. For example, a user-scheduled timer event will be delivered using the main context thread, and therefore cannot complete processing of a retained message.

- **Transport Type** - The Single Receiving Thread feature does not enhance the operation of Broker or SMX transport types. These transport types use somewhat different internal buffer handling. Note that these transport types are technically compatible with the Single Receiving Thread feature, they just don't benefit from it.

8.3 lbm_context_process_events_ex

Most developers of UM applications use a multi-threaded approach to their application design. For example, they typically have one or more application threads, and they create a UM context with **embedded mode**, which creates a separate context thread.

However, there is a model of application design in which a single thread is used for the entire application. In this case, the UM context must be created with Sequential Mode and the application must regularly call the UM event processor API, usually with the `msec` parameter set to zero. In this design, there is no possibility that application code, UM API code, and/or UM context code will be executing concurrently.

The **lbm_context_process_events_ex()** API allows the application to enable specialized optimizations.

8.3.1 Context Lock Reduction

The application can improve performance by suppressing the taking of certain mutex locks within the UM context processing code. This can reduce the latency of delivering received messages to the application, but does so at the expense of thread safety.

This feature is often used in conjunction with the Single Receiving Thread feature.

Warning

> It is very important for the application to ensure that UM code related to a given context cannot be executed concurrently by multiple threads when this feature is used. This includes UM object creation and send-path API functions. I.e. the application may not call a UM message send API by one thread while another thread is calling **lbm_context_process_events_ex()**. However, it is permissible for a context thread callback to call a UM message send API, within the restrictions of the send API being used.

To enable this feature, call **lbm_context_process_events_ex()**, passing in the **lbm_process_events_info_t** structure with the **LBM_PROC_EVENT_EX_FLAG_NO_MAIN_LOOP_MUTEX** bit set in the `flags` field. (Sequential Mode is required for this feature.)

8.3.2 Context Lock Reduction Restrictions

It is very important for applications using this feature to be designed within certain restrictions.

- **Threading** - It is critical that Context Lock Reduction be used *only* if Sequential Mode is used and there is no possibility of concurrent execution of UM code for a given context.

  It is further strongly advised that the same thread be used for all UM execution within a given context. I.e. it is not guaranteed to be safe if the application has multiple threads which can operate on a context, even if the application guarantees that only one thread at a time will execute the UM code.

  Note that if an application maintains two contexts, it is acceptable for a different thread to be used to operate on each context. However, it is not supported to pass UM objects between the threads.

- **No Transport Services Provider (XSP)** - The Context Lock Reduction feature is not compatible with XSP.

- **No Event Queues** - Event queues cannot be used with Context Lock Reduction.

- **No SMX or DBL** - Context Lock Reduction is not compatible with SMX or DBL transports. This is because these transports create independent threads to monitor their respective transport types.

- **Transport LBT-IPC** - Context Lock Reduction was not designed with the IPC transport in mind. By default, IPC creates an independent thread to monitor the shared memory, which is not compatible with Context Lock Reduction. However, in principle, it is possible to specify that the IPC receiver should use sequential mode (see **transport_lbtipc_receiver_operational_mode (context)**), and then write your application to use the same thread to call the context and IPC event processing APIs. However, be aware that the IPC event processing API does not have an extended form, so IPC will simply continue to take the locks it is designed to take.

- **Message Object Retention** - Most traditional uses of Message Object Retention are related to handing a message to an alternate thread for processing. This is not compatible with Context Lock Reduction because the alternate thread is responsible for deleting the message when it is done. This represents two threads making API calls for the same context, which is not allowed for the Context Lock Reduction feature.

  However, there are some use cases where message retention is viable when used with Context Lock Reduction: when a message must be held for *future* processing, and that processing will be done by the same thread.

  For example, a persistent application might use **Explicit Acknowledgments** to delay message acknowledgement until the application completes a handshake with a remote service. As long as it is the same thread which initially receives and retains the message as that which completes the explicit acknowledgement of the message, it is supported to use message retain / message delete.

- **No LBM_SRC_BLOCK** - All forms of UM send message must be done non-blocking (i.e. with **LBM_SRC_NONBLOCK**). This is because of the way UM blocks calls that cannot be completed; the context thread explicitly wakes up the blocked call when appropriate. But if the same thread is being used to run the context (via the process events API) and also sending messages, a blocked send call will never be woken up.

8.3.3 Gettimeofday Reduction

UM's main context loop calls gettimeofday() in strategic places to ensure that its internal timers are processed correctly. However, there is a "polling" model of application design in which Sequential Mode is enabled and the context event processing API is called in a fast loop with the `msec` parameter set to zero. This results in the internal context call to gettimeofday() to happen unnecessarily frequently.

A polling application can improve performance by suppressing the internal context calls to gettimeofday(). This can reduce the latency of delivering received messages to the application.

To enable this feature, call **lbm_context_process_events_ex()**, passing in the **lbm_process_events_info_t** structure with the **LBM_PROC_EVENT_EX_FLAG_USER_TIME** bit set in the `flags` field. In addition, the application must set the `time_val` field in **lbm_process_events_info_t** with the value returned by gettimeofday(). (Sequential Mode is required for this feature.)

Note

> The internal UM timers generally use millisecond precision. Users of the gettimeofday() reduction feature typically design their application to fetch a new value for `time_val` only a few times per millisecond.

8.3.4 Gettimeofday Reduction Restrictions

- **Monotonically Increasing Time** - The application is responsible for ensuring that each call to **lbm_context_process_events_ex()** has a `time_val` field value which is greater than or equal to the previous `time_val`.

8.4 Receive Multiple Datagrams

A UM receiver for UDP-based protocols normally retrieves a single UDP datagram from the socket with each socket read. Setting **multiple_receive_maximum_datagrams (context)** to a value greater than zero directs UM to retrieve up to that many datagrams with each socket read. When receive socket buffers accumulate multiple messages, this feature improves CPU efficiency, which reduces the probability of loss, and also reduces total latency for those buffered datagrams. Note that UM does not need to wait for that many datagrams to be received before processing them; if fewer datagrams are in the socket's receive buffer, only the available datagrams retrieved.

In addition to increasing efficiency, setting **multiple_receive_maximum_datagrams (context)** greater than zero can produce changes in the dynamic behavior across multiple sockets. For example, let's say that a receiver is subscribed to two Transport Sessions, A and B. Let's further say that Transport Session A is sending message relatively quickly and has built up several datagrams in its socket buffer. Further, B is sending slowly. If **multiple_receive_maximum_datagrams (context)** is zero, the two sockets will compete equally for UM's attention. I.e. B's socket will still have a chance to be read after each A datagram is read and processed. However, if **multiple_receive_maximum_datagrams (context)** is 10, then UM can process up to 10 of A's messages before giving B a chance to be read. This is desirable if low message latency is equally important across all Transport Sessions; the efficiency improvement derived by retrieving multiple datagrams with each read operation results in lower overall latency. However, if it is more important to minimize latency of the slower Transport Session's messages, then it would be better to set **multiple_receive_maximum_datagrams (context)** close to or equal to zero.

The **multiple_receive_maximum_datagrams (context)** configuration option defaults to 0 so as to retain previous behavior, but users are encouraged to set this to a value between 2 and 10. Having too large a value during a period of overload can lead to starvation of low-rate Transport Sessions by high-rate Transport Sessions.

8.4.1 Receive Multiple Datagrams Compatibility

The Receive Multiple Datagrams feature is compatible with the following UM features:

- UDP-based transport protocols LBT-RM and LBT-RU.
- MIM (Multicast Immediate Message).
- UDP-based Topic Resolution protocol, both multicast and unicast.
- All language bindings (C, Java, .NET).

8.4.2 Receive Multiple Datagrams Restrictions

The Receive Multiple Datagrams feature is not compatible with the following UM features:

- No Non-UDP Transport Protocols (TCP, IPC, SMX).
- Other TCP-based features (Unicast Immediate Message, Late Join, Persistent Store Recovery, UM Response messages).
- Non-Linux. The recvmmsg() function was introduced into the Linux kernel in version 2.6.33, and support for it was added to glibc in version 2.12.

8.5 Smart Sources

The normal **lbm_src_send()** function (and its Java and .NET equivalents) are very flexible and support the full range of UM's rich feature set. To provide this level of capability, it is necessary to make use of dynamic (malloc/free) memory, and critical section locking (mutex) in the send path. While modern memory managers and thread locks are very efficient, they do introduce some degree of variability of execution time, leading to latency outliers potentially in the millisecond range.

For applications which require even higher speed and very consistent timing, and are able to run within certain constraints, UM has an alternate send feature called Smart Source. This is a highly-optimized send path with no dynamic memory operations or locking; all allocations are done at source creation time, and lockless algorithms are used throughout. To achieve these improvements, Smart Source imposes a number of restrictions (see Smart Sources Restrictions).

The Smart Source feature provides the greatest benefit when used in conjunction with a kernel bypass network driver.

Note

> the Smart Source feature is *not* the same thing as the Zero-Copy Send API feature; see Comparison of Zero Copy and Smart Sources.

One design feature that is central to Smart Sources is the pre-allocation of a fixed number of carefully-sized buffers during source creation. This allows deterministic algorithms to be used for the management of message buffers throughout the send process. To gain the greatest benefit from Smart Sources, the application builds its outgoing messages directly in one of the pre-allocated buffers and submits the buffer to be sent.

To use Smart Sources, a user application typically performs the following steps:

1. Create a context with **lbm_context_create()**, as normal.
2. Create the topic object and the Smart Source with **lbm_src_topic_alloc()** and **lbm_ssrc_create()**, respectively. Use Smart Sources Configuration to pre-allocate the desired number of buffers.
3. Get the desired number of messages buffers with **lbm_ssrc_buff_get()** and initialize them if desired. The application typically constructs outgoing messages directly in these buffers for transmission.
4. Send messages with **lbm_ssrc_send_ex()**. The buffers gotten in the previous step must be used.
5. While most applications manage the message buffers internally, it is also possible to give the buffers back to UM with **lbm_ssrc_buff_put()**, and then getting them again for subsequent sends. Getting and putting messages buffers can simplify application design at the expense of extra overhead.
6. To clean up, delete the Smart Source with **lbm_ssrc_delete()**. It is not necessary to "put" the message buffers back to UM; they will be freed automatically when the Smart Source is deleted.

For details, see the example applications `lbmssrc.c` or `lbmssrc.java`.

Warning

> To avoid the overhead of locking, the Smart Source API functions are not thread-safe. Applications must be written to avoid concurrent calls. In particular, the application is restricted to sending messages on a given Transport Session with one thread. If Smart Source Defensive Checks are enabled, the first call to send a message on a newly-created Transport Session captures the ID of the calling thread. Subsequently, only that thread is allowed to call send for Smart Sources on that Transport Session. For applications which have multiple sending threads, Smart Source topics must be mapped to Transport Sessions carefully such that all of the topics on a given Transport Session are managed by the same sending thread.

Note

> There are no special requirements on the receive side when using Smart Sources. Normal receiving code is used.

8.5.1 Smart Source Message Buffers

When a Smart Source is created, UM pre-allocates a set of user buffers according to the configuration options **smart_src_max_message_length (source)** and **smart_src_user_buffer_count (source)**.

As of UM version 6.12, Smart Source supports UM-level message fragmentation. (See Message Fragmentation and Reassembly for description.) Which is to say that messages larger than the transport's datagram max size can be sent, which the Smart Source will split into multiple datagrams.

For example, an application can configure **smart_src_max_message_length (source)** to be 2000, while the datagram max size is set to 1500 (network MTU size). During operation, the application might send a 500-byte message. This will not require any fragmentation; the message is sent in a single network packet. However, when the application sends a 2000-byte message, the Smart Source will split it into two datagrams. This avoids IP fragmentation. The precise sizes of those datagrams will depend on the space reserved for headers, and is subject to change with different versions of UM.

Another feature available as of UM version 6.12 is the user-specified buffer. This allows an application to send messages larger than the configured **smart_src_max_message_length (source)**. Instead of building the message in a pre-allocated Smart Source buffer, the application must allocate and manage its own user-supplied buffer. To use this feature, the application supplies both a pre-allocated buffer and a user-supplied buffer. The Smart Source will use the supplied pre-allocated buffer as a "work area" for building the datagram with proper headers, and use the user-supplied buffer for message content.

For example:

```
char *ubuffer = malloc(65536); / * Large user-supplied buffer. */
lbm_ssrc_send_ex_info_t info;
info.flags = 0;
char *ss_buffer = NULL; / * Smart Source pre-allocated buffer. */
...
lbm_ssrc_buff_get(ssrc, &ss_buffer, 0); / * Get Smart Source pre-alloc buff. */
.../
* Application puts message data into ubuffer. */
info.flags |= LBM_SSRC_SEND_EX_FLAG_USER_SUPPLIED_BUFFER;
info.usr_supplied_buffer = ubuffer;
lbm_ssrc_send_ex(ssrc, ss_buffer, message_len, 0, &info);
```

Note that sending messages with the user-supplied message buffer is slightly less CPU efficient than using the pre-allocated buffers. But making pre-allocated buffers larger to accommodate occasional large messages can be very wasteful of memory, depending on the counts of user buffers, transmission window buffers, and retention buffers.

UM Fragment Sizes

A traditional source will split application messages into "N" fragments when those messages (plus worst-case header) are greater than the datagram max size. The size of the first "N-1" fragments will be (approximately) the datagram max size.

With Smart Sources, fragmentation is done somewhat differently. Consider as an example a configuration with a datagram max size of 8192 and a Smart Source max message length of 2000. No UM message fragmentation will happen when the application uses the Smart Source pre-allocated buffers to build outgoing messages. However, if a user-supplied buffer is used, the user can send arbitrarily large application message, and the Smart Source will split the message into "N" fragments. But those fragments will be limited in size to the Smart Source max message length of 2000 bytes of application data (plus additional bytes for headers).

This can lead to unexpected inefficiencies. Continuing the above example, suppose case the application sends a 6000-byte message. The Smart Source will spit it into three 2000-byte datagrams. The underlying IP stack will perform IP fragmentation and send each datagram as two packets of 1500 and 500 bytes respectively, for a total of 6 packets. Whereas if the Smart Source max message length were set to 1500, then the message would be split into 4 fragments of 1500 bytes each, and each fragment would fit in a single packet, for a total of 4 packets. (The calculations above were simplified for clarity, but are not accurate because they do not take into consideration headers.)

When a kernel bypass network driver is being used, users will sometimes set the datagram max size to approximately an MTU. In that case, it could easily happen that the Smart Source pre-allocated buffers are *larger* than the datagram max size. In that case, the Smart Source will behave more like a traditional source, splitting the application message into datagrams of (approximately) datagram max size fragments.

8.5.2 Smart Sources and Memory Management

As of UM 6.11, there are new C APIs that give the application greater control over the allocation of memory when Smart Sources are being created. Since creation of a Smart Source pre-allocates buffers used for application message data as well as internal retransmission buffers, an application can override the stock malloc/free to ensure, for example, that memory is local to the CPU core that will be sending messages.

When the application is ready to create the Smart Source, it should set up the configuration option **mem_mgt_callbacks (source)**, which uses the **lbm_mem_mgt_callbacks_t** structure to specify application callback functions.

8.5.3 Smart Sources Configuration

The following configuration options are used to control the creation and operation of Smart Sources:

- **smart_src_max_message_length (source)** - should be set to the maximum expected size for messages sent to on the source.
- **smart_src_user_buffer_count (source)** - number of buffers to be pre-created at Smart Source create time. Deleting a Smart Source also frees these buffers, so applications must not access these buffers after their corresponding Smart Source is deleted.
- **smart_src_retention_buffer_count (source)** - enables Late Join and Off-Transport Recovery (OTR) functionality. Takes the place of the normal late join / OTR options "retransmit_retention_∗". (On the receive side, the normal late join options apply.)
- **transport_lbtrm_smart_src_transmission_window_buffer_count (source)** - size of the LBT-RM transmission window. Takes the place of the normal window options "transport_lbtrm_transmission_window_∗".
- **transport_lbtru_smart_src_transmission_window_buffer_count (source)** - size of the LBT-RU transmission window. Takes the place of the normal window options "transport_lbtru_transmission_window_∗".
- **smart_src_enable_spectrum_channel (source)** - should be set if Spectrum channels will be used. See Smart Sources and Spectrum.
- **smart_src_message_property_int_count (source)** - should be set if Message Properties will be used. See Smart Sources and Message Properties.

The option **smart_src_max_message_length (source)** is used to size the window transmission buffers. This means that the first Smart Source created on the session defines the maximum possible size of user messages for all Smart Sources on the Transport Session. It is not legal to create a subsequent Smart Source on the same Transport Session that has a larger **smart_src_max_message_length (source)**, although smaller values are permissible.

8.5.4 Smart Source Defensive Checks

Ultra Messaging generally includes defensive checks in API functions to verify validity of input parameters. In support of faster operation, deep defensive checks for Smart Sources are optional, and are disabled by default. Users should enable them during application development, and can leave them disabled for production.

To enable deep Smart Source defensive checks, set the environment variable **LBM_SMART_SOURCE_CHECK** to the numeric sum of desired values. Hexadecimal values may be supplied with the "0x" prefix. Each value enables a class of defensive checking:

Numeric Value	Deep Check
1	Send argument checking
2	Thread checking
4	User buffer pointer checking
8	User buffer structure checking
16, 0x10	user message length checking
32, 0x20	application header checking, including Spectrum and Message Properties.

To enable all checking, set the environment variable **LBM_SMART_SOURCE_CHECK** to "0xffffffff".

8.5.5 Smart Sources Restrictions

- **Linux and Windows 64-bit Only** - Smart Sources is only supported on the 64-bit Linux and 64-bit Windows platforms, C and Java APIs.
- **LBT-RM And LBT-RU Sources Only** - Smart Sources can only be created with the LBT-RM and LBT-RU transport types. Non-source-based sends are not supported (MIM, **UIM**, responses).
- **Persistence** - As of UM 6.11, Smart Sources support Persistence, but with some restrictions. See **Smart Sources and Persistence** for details.
- **Spectrum** - As of UM 6.11, Smart Sources support Spectrum, but with some API changes. See Smart Sources and Spectrum for details.
- **Single-threaded Only** - It is the application's responsibility to serialize calls to Smart Source APIs for a given Transport Session. Concurrent sends to different Transport Sessions are permitted.
- **Single-datagram** - Application messages are limited in size to a single datagram. That size is configurable by **transport_lbtrm_datagram_max_size (context)**, which defaults to 8K. (Applications must define the maximum size of messages they intend to send; see the configuration option **smart_src_max_message↩ _length (source)**. This setting must be less than or equal to the **transport_lbtrm_datagram_max_size (context)** minus 44 bytes of overhead.)
- **No Application Headers** - Application messages may not include **application headers**.
- **Limited Message Properties** - Message Properties may be included, but their use has restrictions. See Smart Source Message Properties Usage.
- **No Queuing** - Queuing is not currently supported, although support for ULB is a possibility in the future.
- **No Request** - Sending UM Requests are not currently supported.
- **No Data Rate Limit** - Smart Source data messages are not **rate limited**, although retransmissions are **rate limited**. Care must be taken in designing and provisioning systems to prevent overloading network and host equipment, and overrunning receivers.
- **No Hot Failover** - The Hot Failover feature is not supported by Smart Sources.
- **No Batching** - Neither Implicit Batching nor Explicit Batching are supported by Smart Sources.

  Note

  It is not permitted to mix Smart Source API calls with standard source API calls for a given Transport Session.

8.6 Zero-Copy Send API

This section introduces the use of the zero-copy send API for LBT-RM.

Note

the Zero-Copy Send API feature is *not* the same thing as the Smart Sources feature; see Comparison of Zero Copy and Smart Sources.

The zero-copy send API modifies the **lbm_src_send()** function for sending messages such that the UM library does not copy the user's message data before handing the datagram to the socket layer. These changes reduce CPU overhead and provide a minor reduction in latency. The effects are more pronounced for larger user messages, within the restrictions outlined below.

Application code using the zero-copy send API must call **lbm_src_alloc_msg_buff()** to request a message buffer into which it will build its outgoing message. That function returns a message buffer pointer and also a separate buffer handle. When the application is ready to send the message, it must call **lbm_src_send()**, passing the buffer handle as the message (not the message buffer) and specify the LBM_MSG_BUFF_ALLOC send flag.

Once the message is sent, UM will process the buffer asynchronously. Therefore, the application must not make any further reference to either the buffer or the handle.

8.6.1 Zero-Copy Send Compatibility

The zero-copy send API is compatible with the following UM features:

- C language, Streaming, source-based publishing applications using LBT-RM.
- Messages sent with the zero-copy API can be received by any UM product or daemon. No special restrictions apply to receivers of messages sent with the zero-copy send API.
- Compatible with implicit batching and message flushing.
- Compatible with non-blocking sends and wakeup source event handling.
- Compatible with hardware timestamps (see section High-resolution Timestamps).
- Compatible with UD Acceleration.

8.6.2 Zero-Copy Restrictions

Due to the specialized nature of this feature, there are several restrictions in its use:

- **Languages**: Java and .NET are not supported at this time.
- **Transport**: Sourced-based LBT-RM (multicast) only. Not supported for immediate messages or non-LBT-RM transport types. Note that an application that uses zero-copy sends for certain sources may also have other sources configured for other transport types.
- **Application only**: UM daemons (e.g. UM Router, Stored, etc.) cannot be configured to use the zero-copy API.
- **Streaming only**: Persistence and queuing not supported. Note that an application that uses zero-copy sends for certain sources may also have other sources mapped to Persistence and/or queuing.
- **lbm_src_send() only**: send APIs not supported: **lbm_src_sendv()**, **lbm_src_send_ex()**, **lbm_src_sendv_ex()**, **lbm_hf_src_send()**, **lbm_hf_src_sendv()**, **lbm_hf_src_send_ex()**, **lbm_hf_src_sendv_ex()**, **lbm_send_request()**, **lbm_send_request_ex()**, **lbm_send_response()**, **lbm_multicast_immediate_message()**, **lbm_multicast_immediate_request()**, **lbm_unicast_immediate_message()**, **lbm_unicast_immediate_request()**. Applications may still use these APIs, but not with the zero-copy send feature.
- **Send order**: It is recommended that zero-copy buffers be sent in the same order that they are allocated. A future version may require this.
- **Late join**: not supported. Note that an application that uses zero-copy sends on certain sources may also use late join on other sources.
- **Request/response**: not supported.

- **Metadata**: message properties and application headers are not supported. Note that an application that uses zero-copy sends for messages without metadata may also send messages with metadata using other send APIs, even to the same source.
- **Hot failover**: not supported. Note that an application that uses zero-copy sends for certain sources may use hot failover for other sources.
- **Explicit batching**: not supported. Note that implicit batching is supported. Also note that an application that uses zero-copy sends for certain sources may use explicit batching for other sources.
- **UM Fragmentation**: Not supported. Messages sent zero-copy must fit within a single datagram, as defined by the LBT-RM maximum datagram size. No special restrictions apply to IP fragmentation. Note that an application that uses zero-copy sends for single-datagram messages may also send multi-datagram messages using other send APIs, even to the same source.

8.7 Comparison of Zero Copy and Smart Sources

There are two UM features that are intended to reduce latency and jitter when sending messages:

- Smart Sources
- Zero-Copy Send API

These two features use different approaches to latency and jitter reduction, and are not compatible with each other. There are trade offs explained below, and users seeking latency and/or jitter reduction will sometimes need to try both and empirically measure which is better for their use case.

The zero-copy send API removes a copy of the user's data buffer, as compared to a normal send. For small messages of a few hundred bytes, a malloc and a data copy represent a very small amount of time, so unless your messages are large, the absolute latency reduction is minimal.

The Smart Source has the advantage of eliminating all mallocs and frees from the send path. In addition, all thread locking is eliminated. This essentially removes all sources of jitter from the UM send path. However, because of the approach taken, sending to a Smart Source is more restrictive than sending with the zero-copy API.

In general, Informatica recommends Smart Sources to achieve the maximum reduction in jitter. For example, the zero-copy send API supports the use of batching to combine multiple messages into a single network datagram. Batching can be essential to achieve high throughputs. Some application designers may determine that the throughput advantages of zero-copy with batching outweigh the jitter advantages of Smart Sources.

See the sections Zero-Copy Send API and Smart Sources for details of their restrictions.

8.8 UM Daemons as Windows Services

TBD

Chapter 9

Manpage for SRS

Help for the Stateful Resolution Service (SRS) command line can be obtained by entering "SRS -h". Brief help for the SRS configuration file can be obtained by entering "SRS -x", which prints the XSD contents.

For more information on TCP-based TR, see TCP-Based Topic Resolution Details.

For more information on Topic Resolution general, see Topic Resolution Description.

9.1 SRS Command Line

```
Usage: SRS [options]
Available options:
  -c, --config=FILE        use SRS configuration file FILE
                           Example:  '-c srs.xml'
  -d, --dump               dump the user configuration to stdout and exit
  -D, --Debug=PATH:MASK    set debug PATH and MASK
  -h, --help               display this help message and exit
  -j, --java               print Java properties to the SRS log file (-j -j = print
      more Java properties)
  -v, --validate           validate config file and exit
  -x, --xsd                dump the configuration XSD to stdout and exit
```

Description

TCP-based resolver services for UM messaging products are provided by SRS.

The `-c` option supplies the name of a file containing the configuration for the SRS. This file must be in XML format, and is validated against the XSD file shown below.

The `-d` option prints to standard out the full SRS configuration. After printing, the SRS exits.

The `-D` option sets enables debugging output. This output is intended primarily for Informatica Support, not end-user consumption.

The `-h` option prints the above usage page. After printing, the SRS exits.

The `-x` option prints the XSD which is used to validate the configuration file. After printing, the SRS exits.

The `-v` option causes the SRS to simply validate the configuration file and print any errors it finds. After printing, the SRS exits.

The `-j` option prints Java properties to the SRS log file. It can be repeated ('-j -j') to increase the output. This output is intended primarily for Informatica Support, not end-user consumption.

Exit Status

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

Chapter 10

SRS Configuration File

The SRS configuration file must start with this line:

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

After that, the '<**um-srs**>' element contains the rest of the configuration.

Here is a sample short configuration:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <log type="file" frequency="hourly" size="10" max-history="10"
         total-size-cap="10000" compression="zip">SRS.log</log>
    <pid-file>SRS.pid</pid-file>
  </daemon>
  <srs>
    <interface>localhost</interface>
    <port>27000</port>
    <state-lifetime>3600</state-lifetime>
  </srs>
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <default>2000</default>
      <config-opts>20000</config-opts>
      <internal-config-opts>0</internal-config-opts>
    </publishing-interval>
    <lbm-attributes>
      <option name="transport" scope="source" value="tcp" />
      <option name="transport_tcp_port_low" scope="context" value="14381" />
      <option name="transport_tcp_port_high" scope="context" value="15381" />
      <option name="transport_tcp_interface" scope="source" value="10.29.3.0/24" />
    </lbm-attributes>
  </daemon-monitor>
</um-srs>
```

TBD - turn off command and control?

10.1 SRS Configuration Elements

10.1.1 SRS Element "<um-srs>"

Container element which holds the SRS configuration. Also defines the version of the configuration format used by the file.

- **Children:** <daemon>, <srs>, <debug-monitor>, <daemon-monitor>

XML Attributes:

Attribute	Description	Valid Values	Default Value
version	Version number of user's configuration file.	nonEmptyString	**1.0**

Example:

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

10.1.2 SRS Element "<daemon-monitor>"

Contains elements which configure the SRS monitoring capability. This feature is used to monitor the SRS's health and performance. It can also be useful to monitor activity in the entire Topic Resolution Domain.

See child elements for details.

- **Cardinality:** 0 .. 1
- **Parent:** <um-srs>
- **Children:** <publishing-interval>, <publish-connection-events>, <lbm-attributes>

XML Attributes:

Attribute	Description	Valid Values	Default Value
topic	Set the name of the topic on which the SRS publishes its daemon stats.	nonEmptyString	(no default; must be specified)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.3 SRS Element "<lbm-attributes>"

Container element containing any number of <option> elements. Each <option> element supplies a UM configuration option to the UM context that the SRS creates to publish daemon stats. Any number of <option> elements can be supplied in the <lbm-attributes> container element.

See SRS Daemon Statistics for more information on daemon statistics.

- **Cardinality:** 0 .. 1
- **Parent:** <daemon-monitor>
- **Children:** <option>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <lbm-attributes>
      ...
    </lbm-attributes>
  </daemon-monitor>
  ...
</um-srs>
```

10.1.4 SRS Element "<option>"

Supplies a UM configuration option to the UM context that the SRS creates to publish daemon stats. Any number of <option> elements can be supplied in the <lbm-attributes> container element.

See SRS Daemon Statistics for more information on daemon statistics.

- **Cardinality:** 0 .. unbounded
- **Parent:** <lbm-attributes>

XML Attributes:

Attribute	Description	Valid Values	Default Value
scope	Scope for the UM configuration option being set. One of: **`context`** **`source`** **`receiver`** (The normal UM scopes `wildcard_receiver`, `event_queue`, and `hfx` are not applicable to the SRS monitor context.)	nonEmptyString	(no default; must be specified)
name	Name of UM configuration option being set.	nonEmptyString	(no default; must be specified)
value	Value of UM configuration option being set.	nonEmptyString	(no default; must be specified)

Example: (SRS publishes monitoring stats using LBT-RU transport)

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

```
  <daemon-monitor topic="SrsDaemonStats">
    <lbm-attributes>
      <option scope="source" name="transport" value="tcp"/>
      ...
    </lbm-attributes>
  </daemon-monitor>
  ...
</um-srs>
```

10.1.5 SRS Element "<publish-connection-events>"

Controls whether the SRS reports connection-oriented events from UM contexts as part of the daemon stats. Can be set to `true` or `false`. See SRS Daemon Statistics for more information.

- **Cardinality:** 0 .. 1
- **Parent:** <daemon-monitor>
- **Default Value:** false

Example: (SRS includes connection-oriented events in monitoring stats)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publish-connection-events>true</publish-connection-events>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.6 SRS Element "<publishing-interval>"

Set how often the SRS publishes its daemon stats. See SRS Daemon Statistics for more information. The child elements set the intervals for each class of monitoring data. For any class of data omitted, the <default> element sets the interval.

- **Cardinality:** 0 .. 1
- **Parent:** <daemon-monitor>
- **Children:** <default>, <srs-stats>, <um-client-stats>, <connection-events>, <srs-error-stats>, <um-client-error-stats>, <config-opts>, <internal-config-opts>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.7 SRS Element "<internal-config-opts>"

Sets how often (in milliseconds) the SRS publishes certain internal configuration data. These data are primarily of interest to Informatica Support. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** 10

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <internal-config-opts>10000</internal-config-opts>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.8 SRS Element "<config-opts>"

Sets how often (in milliseconds) the SRS publishes its configuration data. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <config-opts>10000</config-opts>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.9 SRS Element "<um-client-error-stats>"

Sets how often (in milliseconds) the SRS publishes statistics related to internal client-facing software errors. These statistics are primarily of interest to Informatica Support. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <um-client-error-stats>10000</um-client-error-stats>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.10 SRS Element "<srs-error-stats>"

Sets how often (in milliseconds) the SRS publishes statistics related to internal SRS software errors. These statistics are primarily of interest to Informatica Support. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <srs-error-stats>10000</srs-error-stats>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.11 SRS Element "<connection-events>"

Sets how often (in milliseconds) the SRS publishes client connect and disconnect events. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <connection-events>10000</connection-events>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.12 SRS Element "<um-client-stats>"

Sets how often (in milliseconds) the SRS publishes statistics related to Topic Resolution clients. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <um-client-stats>10000</um-client-stats>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.13 SRS Element "<srs-stats>"

Sets how often (in milliseconds) the SRS publishes internal SRS operational statistics. The value zero disables publishing that class of daemon stats.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** Value supplied by <default>.

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <srs-stats>10000</srs-stats>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.14 SRS Element "<default>"

Sets how often (in milliseconds) the SRS publishes those classes of daemon stats which are not explicitly set by other elements. The value zero disables publishing that class of daemon stats. See SRS Element "<publishing-interval>" for the classes.

- **Cardinality:** 0 .. 1
- **Parent:** <publishing-interval>
- **Default Value:** 10

Example: (SRS publishes monitoring stats every 10 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon-monitor topic="SrsDaemonStats">
    <publishing-interval>
      <default>10000</default>
      ...
    </publishing-interval>
    ...
  </daemon-monitor>
  ...
</um-srs>
```

10.1.15 SRS Element "<debug-monitor>"

Contains elements which configure the optional web-based debug monitor for the SRS. The debug monitor is primarily for use by Informatica support, and is not intended for end users. Unless otherwise instructed by Informatica support, users should not enable the debug monitor.

This is NOT related to monitoring the SRS health and performance. See <daemon-monitor>.

- **Cardinality:** 0 .. 1

- **Parent:** <um-srs>
- **Children:** <interface>, <port>, <ping-interval>, <enabled>

Example:

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

10.1.16 SRS Element "<enabled>"

Controls whether the debug monitor is active. Can be set to `true` or `false`. See <debug-monitor>.

- **Cardinality:** 0 .. 1
- **Parent:** <debug-monitor>
- **Default Value:** false

Example: (disable debug-monitor explicitly)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <debug-monitor>
    <enabled>false</enabled>
    ...
  </debug-monitor>
  ...
</um-srs>
```

10.1.17 SRS Element "<ping-interval>"

Controls the period (in milliseconds) at which SRS internal statistics are sampled and made available to the debug monitor. Valid values: 0 - 60000 (1 minute). The value zero disables this sampling.

- **Cardinality:** 0 .. 1
- **Parent:** <debug-monitor>
- **Default Value:** 60000 (1 minute)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <debug-monitor>
    <ping-interval>10000</ping-interval>
    ...
  </debug-monitor>
  ...
</um-srs>
```

10.1.18 SRS Element "<port>"

Supplies network port to bind the socket required by the parent element. This is the port that a UM context should use when TCP-based TR is configured with the option **resolver_service (context)**. The value contained within the <port>...</port> is an integer between 1 and 65535.

- **Cardinality:** 0 .. 1
- **Parent:** <debug-monitor>, <srs>
- **Default Value:** 27000

Example: (UM clients use port 12000 with **resolver_service (context)** option)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <port>12000</port>
    ...
  </srs>
  ...
</um-srs>
```

10.1.19 SRS Element "<interface>"

Specifies the network interface to bind the socket required by the parent element.

For the <srs> element, this is the IP address that a UM context should use when TCP-based TR is configured with the option **resolver_service (context)**. The value contained within the <interface>...</interface> can be a dotted-decimal IP address or a DNS host name.

For the <debug-monitor> element, this is the host for the URL that a web browser should use to display the debug monitor page.

- **Cardinality:** 0 .. 1
- **Parent:** <debug-monitor>, <srs>
- **Default Value:** localhost

Example: (UM clients use 10.12.34.56 with **resolver_service (context)** option)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <interface>10.12.34.56</interface>
    ...
  </srs>
  ...
</um-srs>
```

10.1.20 SRS Element "<srs>"

Defines network and operational settings of the SRS service.

- **Cardinality:** 0 .. 1
- **Parent:** <um-srs>
- **Children:** <interface>, <port>, <state-lifetime>, <otidmap>, <clientactor>

Example:

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

10.1.21 SRS Element "<clientactor>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <srs>
- **Children:** <request-stream-max-msg-count>, <source-info-queue-service-interval>, <batch-frame-max-record-count>, <batch-frame-max-datagram-size>

10.1.22 SRS Element "<batch-frame-max-datagram-size>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <clientactor>

10.1.23 SRS Element "<batch-frame-max-record-count>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <clientactor>

10.1.24 SRS Element "<source-info-queue-service-interval>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <clientactor>

10.1.25 SRS Element "<request-stream-max-msg-count>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <clientactor>

10.1.26 SRS Element "<otidmap>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <srs>
- **Children:** <shards>, <async-receiver-distribution>

10.1.27 SRS Element "<async-receiver-distribution>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <otidmap>

10.1.28 SRS Element "<shards>"

This is for Informatica internal use only. Do not set unless directed to do so by Informatica Support.

- **Cardinality:** 0 .. 1
- **Parent:** <otidmap>

10.1.29 SRS Element "<state-lifetime>"

Sets the value (in seconds) of the client state lifetime.

If a client context loses connection with SRS, the sources contained by that context will be remembered by the SRS for a limited amount of time: the *state lifetime*. If the context does not re-connect within that time, the SRS implicitly deletes all of the sources owned by that lost context. Those deletions will be shared with all connected client contexts.

Zero is a special value which disables the timing of sources of disconnected contexts. I.e. with zero, the sources from a lost context are never deleted. This is generally not recommended as it can lead to unlimited memory growth in both the SRS and in client contexts.

See srsstatelifetime for more information.

- **Cardinality:** 0 .. 1
- **Parent:** <srs>
- **Default Value:** 86400 (24 hours)

Example: (SRS deletes a lost context's sources after 120 seconds)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <srs>
    <state-lifetime>120</state-lifetime>
    ...
  </srs>
  ...
</um-srs>
```

10.1.30 SRS Element "<daemon>"

Contains elements which define logging behavior and sets a file name for the service's Process ID.

See child elements for details.

- **Cardinality:** 0 .. 1
- **Parent:** <um-srs>
- **Children:** <log>, <pid-file>

Example:

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

10.1.31 SRS Element "<pid-file>"

Supplies the desired name of file in which the SRS writes its Process ID (PID).

- **Cardinality:** 0 .. 1
- **Parent:** <daemon>

Example: (SRS writes process ID to "srs_pid.txt" file)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <pid-file>srs_pid.txt</pid-file>
    ...
  </daemon>
  ...
</um-srs>
```

10.1.32 SRS Element "<log>"

Configures SRS logging behavior. The value contained within the `<log>...</log>` is a file name, but is only used if the "type" attribute is set to "file".

When the `type` attribute is set to `"file"`, the SRS supports "rolling" the log file, which consists a series of files over time so that no one file grows too large.

- **Cardinality:** 0 .. 1
- **Parent:** <daemon>

XML Attributes:

Attribute	Description	Valid Values	Default Value
type	Where to write log messages.	**file** - Write log messages to a file. **console** - Write log messages to standard output.	**console**
frequency	Time-frame by which to roll the log file.	**disable** - Do not roll the log file based on time. **daily** - Roll the log file at midnight. **hourly** - Roll the log file each hour.	disable
size	Size (in MB, i.e. 2**20, or 1,048,576) of current log file at which it is rolled. Specify 0 to disable rolling by log file size.	positiveInteger	10 (10,485,760 bytes)
max-history	Number of rolled log files at which the oldest file is deleted when the current log file is rolled.	positiveInteger	10

Attribute	Description	Valid Values	Default Value
total-size-cap	Total disk space consumed (in MB, i.e. 2**20, or 1,048,576) by rolled log files at which the oldest file is deleted to make room for the next log roll.	positiveInteger	1000 (1,048,576,000 bytes)
compression	Enables compression for rolled log files.	**none** - Do not compress log files. **zip** - Compress log files using "zip" format. **gzip** - Compress log files using "gzip" format.	**none**

Example 1: (write log messages to standard out)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <log type="console"/>
    ...
  </daemon>
  ...
</um-srs>
```

Example 2: (write log messages to "srs.log" file)

```
<?xml version="1.0" encoding="UTF-8" ?>
<um-srs version="1.0">
  <daemon>
    <log type="file" frequency="daily">srs.log</log>
    ...
  </daemon>
  ...
</um-srs>
```

10.2 SRS XSD file

The XSD file is used to validate the user's configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="um-srs" type="um-srsType"/>

  <!-- Custom types and restrictions -->
  <xs:simpleType name="nonEmptyString">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="logTypeEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="file"/>
      <xs:enumeration value="console"/>
    </xs:restriction>
  </xs:simpleType>
```

```
<xs:simpleType name="logFrequencyEnumeration">
  <xs:restriction base="xs:string">
    <xs:enumeration value="disable"/>
    <xs:enumeration value="daily"/>
    <xs:enumeration value="hourly"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="compressionEnumeration">
  <xs:restriction base="xs:string">
    <xs:enumeration value="none"/>
    <xs:enumeration value="zip"/>
    <xs:enumeration value="gzip"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="portInteger">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="65535"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="booleanEnumeration">
  <xs:restriction base="xs:string">
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
  </xs:restriction>
</xs:simpleType>

<!-- Acceptable values for publishingIntervalLong type are 0 or >= 200 -->
<xs:simpleType name="publishingIntervalLong">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:long">
        <xs:pattern value="0"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:long">
        <xs:minInclusive value="200"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>


<xs:complexType name="logType" >
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute type="logTypeEnumeration" name="type" use="required"/>
      <xs:attribute type="logFrequencyEnumeration" name="frequency"/>
      <xs:attribute type="xs:positiveInteger" name="size"/>
      <xs:attribute type="xs:positiveInteger" name="max-history"/>
      <xs:attribute type="xs:positiveInteger" name="total-size-cap"/>
      <xs:attribute type="compressionEnumeration" name="compression"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="otidmapType">
  <xs:all>
    <xs:element type="xs:positiveInteger" name="shards" minOccurs="0"
        maxOccurs="1"/>
    <xs:element type="booleanEnumeration" name="async-receiver-distribution"
        minOccurs="0" maxOccurs="1"/>
  </xs:all>
```

```
</xs:complexType>
<xs:complexType name="clientActorType">
  <xs:all>
    <xs:element type="xs:positiveInteger" name="request-stream-max-msg-count"
        minOccurs="0" maxOccurs="1"/>
    <xs:element type="xs:positiveInteger"
        name="source-info-queue-service-interval" minOccurs="0" maxOccurs="1"/>
    <xs:element type="xs:positiveInteger" name="batch-frame-max-record-count"
        minOccurs="0" maxOccurs="1"/>
    <xs:element type="xs:positiveInteger" name="batch-frame-max-datagram-size"
        minOccurs="0" maxOccurs="1"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="daemonType" >
  <xs:all>
    <xs:element type="logType" name="log" minOccurs="0" maxOccurs="1"/>
    <xs:element type="nonEmptyString" name="pid-file" minOccurs="0" maxOccurs="1"/
        >
  </xs:all>
</xs:complexType>
<xs:complexType name="debugMonitorType">
  <xs:all>
    <xs:element type="nonEmptyString" name="interface" minOccurs="0"
        maxOccurs="1"/>
    <xs:element type="portInteger" name="port" minOccurs="0" maxOccurs="1"/>
    <xs:element type="xs:positiveInteger" name="ping-interval" minOccurs="0"
        maxOccurs="1"/>
    <xs:element type="booleanEnumeration" name="enabled" minOccurs="0"
        maxOccurs="1"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="lbmOptionType" >
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute type="nonEmptyString" name="scope" use="required"/>
      <xs:attribute type="nonEmptyString" name="name" use="required"/>
      <xs:attribute type="nonEmptyString" name="value" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="lbmAttributesType">
  <xs:sequence>
    <xs:element type="lbmOptionType" name="option" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="daemonMonitorType" mixed="true">
  <xs:all>
    <xs:element type="publishingIntervalType" name="publishing-interval"
        minOccurs="0" maxOccurs="1"/>
    <xs:element type="booleanEnumeration" name="publish-connection-events"
        minOccurs="0" maxOccurs="1"/>
    <xs:element type="lbmAttributesType" name="lbm-attributes" minOccurs="0"
        maxOccurs="1"/>
  </xs:all>
  <xs:attribute type="nonEmptyString" name="topic"/>
</xs:complexType>
<xs:complexType name="publishingIntervalType">
  <xs:all>
    <xs:element type="publishingIntervalLong" name="default" minOccurs="0"
        maxOccurs="1"/>
    <xs:element type="publishingIntervalLong" name="srs-stats" minOccurs="0"
        maxOccurs="1"/>
```

```
      <xs:element type="publishingIntervalLong" name="um-client-stats"
          minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="connection-events"
          minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="srs-error-stats"
          minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="um-client-error-stats"
          minOccurs="0" maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="config-opts" minOccurs="0"
          maxOccurs="1"/>
      <xs:element type="publishingIntervalLong" name="internal-config-opts"
          minOccurs="0" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="um-srsType">
    <xs:all>
      <xs:element type="daemonType" name="daemon" minOccurs="0" maxOccurs="1"/>
      <xs:element type="srsType" name="srs" minOccurs="0" maxOccurs="1"/>
      <xs:element type="debugMonitorType" name="debug-monitor" minOccurs="0"
          maxOccurs="1"/>
      <xs:element type="daemonMonitorType" name="daemon-monitor" minOccurs="0"
          maxOccurs="1"/>
    </xs:all>
    <xs:attribute type="nonEmptyString" name="version"/>
  </xs:complexType>
  <xs:complexType name="srsType">
    <xs:all>
      <xs:element type="nonEmptyString" name="interface" minOccurs="0"
          maxOccurs="1"/>
      <xs:element type="portInteger" name="port" minOccurs="0" maxOccurs="1"/>
      <xs:element type="xs:integer" name="state-lifetime" minOccurs="0"
          maxOccurs="1"/>
      <xs:element type="otidmapType" name="otidmap" minOccurs="0" maxOccurs="1"/>
      <xs:element type="clientActorType" name="clientactor" minOccurs="0"
          maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
</xs:schema>
```

Chapter 11

SRS Daemon Statistics

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

The SRS Daemon Statistics are published in the form of JSON messages. These are ASCII text messages which represent internal SRS data structures containing statistical and configuration information.

The following sub-sections describe the content of the messages. Note that while the sample messages shown are "beautified" (whitespace inserted for readability), a receiver of these messages should make no assumption about the presence or absence of whitespace.

The message types are:

- Message Type: SRS_STATS
- Message Type: SRS_ERROR_STATS
- Message Type: UM_CLIENT_STATS
- Message Type: UM_CLIENT_ERROR_STATS
- Message Type: CONNECTION_EVENTS
- Message Type: CONFIG_OPTS
- Message Type: INTERNAL_CONFIG_OPTS
- Request Type: REPORT_SRS_VERSION
- Request Type: REPORT_MONITOR_INFO
- Request Type: SET_PUBLISHING_INTERVAL

11.1 Message Type: SRS_STATS

Message type SRS_STATS contains information about the overall state of the SRS service.


```
{
  "monitorInfoCategory": "SRS_STATS",
  "stats": [
    {
      "name": "clients.inactive.SIR.count",
      "value": 0
    },
```

```
    {
      "name": "clients.next.client.ID",
      "value": 17
    },
    {
      "name": "clients.active.SIR.count",
      "value": 1
    },
    {
      "name": "clients.max.concurrent.connections.count",
      "value": 3
    },
    {
      "name": "active.clients.count",
      "value": 2
    },
    {
      "name": "clients.disconnects.count",
      "value": 14
    },
    {
      "name": "clients.connects.count",
      "value": 16
    }
  ]
}
```

Overall structure of message:

<table>
<tr><th>Field</th><th colspan="2">Description</th></tr>
<tr><td>monitorInfoCategory</td><td colspan="2">Message type.</td></tr>
<tr><td>stats</td><td>Array of sub-structures, one per statistic.</td><td><table><tr><td>name</td><td>Name of statistic.</td></tr><tr><td>value</td><td>Value of statistic.</td></tr></table></td></tr>
</table>

Meaning of each statistic:

Statistic	Description
clients.inactive.SIR.count	Message type.
clients.next.client.ID	blah
clients.active.SIR.count	blah
clients.max.concurrent.connections.count	blah
active.clients.count	blah
clients.disconnects.count	blah
clients.connects.count	blah

All of the above statistics are included in a snapshot. Only the changed statistics are included during a periodic update.

11.2 Message Type: SRS_ERROR_STATS

11.3 Message Type: UM_CLIENT_STATS

11.4 Message Type: UM_CLIENT_ERROR_STATS

11.5 Message Type: CONNECTION_EVENTS

11.6 Message Type: CONFIG_OPTS

11.7 Message Type: INTERNAL_CONFIG_OPTS

11.8 Request Type: REPORT_SRS_VERSION

11.9 Request Type: REPORT_MONITOR_INFO

11.10 Request Type: SET_PUBLISHING_INTERVAL

Chapter 12

Manpage for lbmrd

Help for the lbmrd command line can be obtained by entering "lbmrd -h". Help for the lbmrd configuration file can be obtained by entering "lbmrd -d".

For more information on UDP-based TR, see UDP-Based Topic Resolution Details.

For more information on Topic Resolution general, see Topic Resolution Description.

12.1 lbmrd Command Line

```
Usage: lbmrd [options] [config-file]
Available options:
  -a, --activity=IVL    interval between client activity checks (in
      milliseconds)(default 60000)
  -d, --dump-dtd        dump the configuration DTD to stdout and exit
  -h, --help            display this help and exit
  -i, --interface=ADDR  listen for unicast topic resolution messages on interface
      ADDR
                        ADDR accepts CIDR eg:10.0.0.0/8, Quoted device name
                            eg:"eth0", DNS Host name eg:host.mydomain.com/24.
  -L, --logfile=FILE    use FILE as the log file
  -p, --port=PORT       use UDP port PORT for topic resolution messages (default
      15380)
  -t, --ttl=TTL         use client time-to-live of TTL seconds (default 60)
  -r, --rcv-buf=SIZE    set the receive buffer to SIZE bytes.
  -s, --snd-buf=SIZE    set the send buffer to SIZE bytes.
  -v, --validate        validate config-file then exit
```

Description

Unicast UDP-based 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. See **Specifying Interfaces** for methods of specifying the interface.

The `-a` and `-t` (or `--activity` and `--ttl`) options interact to detect and remove "dead" clients, i.e., U↩
MS/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.

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.

`config-file` is the XML configuration file. It 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.

Chapter 13

lbmrd Configuration File

The lbmrd configuration file must start with this line:

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

After that, the '<**lbmrd**>' element contains the rest of the configuration.

Note

The configuration file must contain a '<**domains**>' element and a '<**transformations**>' element (and their contents), even if there is no NAT. See Dummy lbmrd Configuration File. The '<**daemon**>' element and its contents are optional.

13.1 lbmrd Configuration Elements

13.1.1 LBMRD Element "<lbmrd>"

Container element which holds the lbmrd configuration. Also defines the version of the configuration format used by the file.

- **Children:** <daemon>, <domains>, <transformations>

XML Attributes:

Attribute	Description	Valid Values	Default Value
version	Version number of user's configuration file.	**1.0** - Initial version	**1.0**

Example:

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

13.1.2 LBMRD Element "<transformations>"

Container element for definitions of NAT translations applied to TIRs. Translations are used to help lbmrd know how to modify source advertisements when Network Address Translation (NAT) is being used.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <lbmrd>
- **Children:** <transform>

Example:

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

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.3 LBMRD Element "<transform>"

Defines a set of transformation tuples. Each tuple applies to a TIR sent from a specific network domain (specified using the `source` attribute), and destined for a specific network domain (specified using the `destination` attribute). The `source` and `destination` attributes must specify network domain names as defined by the <domain> elements.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <transformations>
- **Children:** <rule>

XML Attributes:

Attribute	Description	Valid Values	Default Value
source	Name of source network domain, defined in <domain>.	IDREF	(no default; must be specified)
destination	Name of receiver network domain, defined in <domain>.	IDREF	(no default; must be specified)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      ...
    </transform>
    ...
```

```
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.4 LBMRD Element "<rule>"

Container for a transformation rule which maps one address and port to another.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <transform>
- **Children:** <match>, <replace>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        ...
      </rule>
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.5 LBMRD Element "<replace>"

Defines the address and port which are to replace those matched in the TIR originating from a UM context within the `source` network (as specified by <transform>), and being delivered to contexts within the `destination` network.

- **Parent:** <rule>

XML Attributes:

Attribute	Description	Valid Values	Default Value
address	IP address within a TIR. Address must be specified only in dotted-decimal and refer to a specific host. For LBMRD Element "<match>", the IP address should be within the network specified by <transform> `source` attribute. For LBMRD Element "<replace>", the IP address should be within the network specified by <transform> `destination` attribute.	string	(no default; must be specified)

Attribute	Description	Valid Values	Default Value
port	Port number to match or replace. To match any port, use value ∗. To replace with same port as matched, use value ∗.	string	∗

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        <match address="10.1.1.50" port="*"/>
        <replace address="192.168.1.1" port="*"/>
        ...
      </rule>
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.6 LBMRD Element "<match>"

Defines the address and port to match within a TIR originating from a UM context within the `source` network (as specified by <transform>), and being delivered to contexts within the `destination` network.

- **Parent:** <rule>

XML Attributes:

Attribute	Description	Valid Values	Default Value
address	IP address within a TIR. Address must be specified only in dotted-decimal and refer to a specific host. For LBMRD Element "<match>", the IP address should be within the network specified by <transform> `source` attribute. For LBMRD Element "<replace>", the IP address should be within the network specified by <transform> `destination` attribute.	string	(no default; must be specified)
port	Port number to match or replace. To match any port, use value ∗. To replace with same port as matched, use value ∗.	string	∗

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <transformations>
    <transform source="Net-NYC" destination="Net-NJC">
      <rule>
        <match address="10.1.1.50" port="*"/>
```

```
        <replace address="192.168.1.1" port="*"/>
        ...
      </rule>
      ...
    </transform>
    ...
  </transformations>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.7 LBMRD Element "<domains>"

Container element for definitions of network domains. Network domains are used to help lbmrd recognize networks and/or subnetworks which connect via Network Address Translation (NAT).

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <lbmrd>
- **Children:** <domain>

Example:

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

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.8 LBMRD Element "<domain>"

Defines a network domain. The domain must be given a unique name via the `name` attribute. This name is referenced in <transform> elements. The `<domain>` element contains one or more <network> elements.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <domains>
- **Children:** <network>

XML Attributes:

Attribute	Description	Valid Values	Default Value
name	Unique name assigned to the defined network.	ID	(no default; must be specified)

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <domains>
    <domain name="Net-NYC">
      ...
    </domain>
    <domain name="Net-NJC">
      ...
    </domain>
    ...
  </domains>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.9 LBMRD Element "<network>"

Defines a single network specification which is to be considered part of the enclosing <domain> element. The network specification must contain either an IP address, or a network specification in `CIDR notation`. DNS host names are not supported in the lbmrd configuration file.

See Network Address Translation (NAT) for more information on NAT.

- **Parent:** <domain>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <domains>
    <domain name="Net-NYC">
      <network>10.1.0.0/16</network>
      ...
    </domain>
    <domain name="Net-NJC">
      <network>192.168.1/24</network>
      ...
    </domain>
    ...
  </domains>
  ...
</lbmrd>
```

For a full example of an lbmrd NAT configuration, see Example NAT Configuration.

13.1.10 LBMRD Element "<daemon>"

Container element for configuration related to the overall lbmrd process.

- **Cardinality:** 0 .. 1
- **Parent:** <lbmrd>
- **Children:** <activity>, <interface>, <port>, <ttl>, <log>, <resolver_unicast_receiver_socket_buffer>, <resolver_unicast_send_socket_buffer>

Example:

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

13.1.11 LBMRD Element "<resolver_unicast_send_socket_buffer>"

Sets the send-side socket buffer size (in bytes).

- **Parent:** <daemon>
- **Default Value:** 1048576

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <resolver_unicast_send_socket_buffer>
      1048576
    </resolver_unicast_send_socket_buffer>
    ...
  </daemon>
  ...
</lbmrd>
```

13.1.12 LBMRD Element "<resolver_unicast_receiver_socket_buffer>"

Sets the receive-side socket buffer size (in bytes).

- **Parent:** <daemon>
- **Default Value:** 1048576

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <resolver_unicast_receiver_socket_buffer>
      1048576
    </resolver_unicast_receiver_socket_buffer>
    ...
  </daemon>
  ...
</lbmrd>
```

13.1.13 LBMRD Element "<log>"

Specifies the file name used for lbmrd logging.

- **Parent:** <daemon>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <log>lbmrd.log</log>
    ...
  </daemon>
  ...
</lbmrd>
```

13.1.14 LBMRD Element "<ttl>"

Interval (in milliseconds) between keep alive checks between the lbmrd and the UM contexts.

- **Parent:** <daemon>
- **Default Value:** 60000

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <ttl>60000</ttl>
    ...
  </daemon>
  ...
</lbmrd>
```

13.1.15 LBMRD Element "<port>"

Supplies network port to bind the socket for receiving TR traffic from UM contexts. This is the port that a UM context should use when TCP-based TR is configured with the option **resolver_unicast_daemon (context)**. The value contained within the `<port>...</port>` is an integer between 1 and 65535.

- **Parent:** <daemon>
- **Default Value:** 15380

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <port>15380</interface>
    ...
  </daemon>
  ...
</lbmrd>
```

13.1.16 LBMRD Element "<interface>"

Specifies the network interface to bind the socket for receiving TR traffic from UM contexts. This is the IP address that a UM context should use when Unicast UDP-based TR is configured with the option **resolver_unicast_daemon (context)**. See **Specifying Interfaces** for methods of specifying the interface within `<interface>...</interface>`.

- **Parent:** <daemon>

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <interface>10.1.1.50</interface>
    ...
  </daemon>
  ...
</lbmrd>
```

13.1.17 LBMRD Element "<activity>"

Interval between client activity checks (in milliseconds)

- **Parent:** <daemon>
- **Default Value:** 60000

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<lbmrd version="1.0">
  <daemon>
    <activity>60000</activity>
    ...
  </daemon>
  ...
</lbmrd>
```

13.2 Dummy lbmrd Configuration File

If no NAT is present, and it is desired to use the XML configuration file for it's '<**daemon**>' contents, a "dummy" NAT configuration should be used.

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

```
  </daemon>
  <domains>
    <domain name="dummy">
      <network>0.0.0.0/32</network>
    </domain>
  </domains>
  <transformations>
    <transform source="dummy" destination="dummy">
      <rule>
        <match address="0.0.0.0" port="0"/>
        <replace address="0.0.0.0" port="0"/>
      </rule>
    </transform>
  </transformations>
</lbmrd>
```

13.3 Lbmrd DTD file

The DTD file is used to validate the user's configuration file.

```
<!ELEMENT lbmrd (daemon?, domains, transformations)>
<!ATTLIST lbmrd
          version (1.0) #REQUIRED
>
<!ELEMENT daemon
    (activity|interface|port|ttl|log|resolver_unicast_receiver_socket_buffer|resolver_unicast_send
<!ELEMENT activity (#PCDATA) >
<!ELEMENT interface (#PCDATA) >
<!ELEMENT port (#PCDATA) >
<!ELEMENT ttl (#PCDATA) >
<!ELEMENT log (#PCDATA) >
<!ELEMENT resolver_unicast_receiver_socket_buffer (#PCDATA) >
<!ELEMENT resolver_unicast_send_socket_buffer (#PCDATA) >
<!ELEMENT domains (domain+)>
<!ELEMENT domain (network+)>
<!ATTLIST domain name ID #REQUIRED>
<!ELEMENT network ( #PCDATA )>
<!ELEMENT transformations ( transform+ )>
<!ELEMENT transform ( rule+ )>
<!ATTLIST transform
          source IDREF #REQUIRED
          destination IDREF #REQUIRED
>
<!ELEMENT rule ( match, replace )>
<!ELEMENT match EMPTY>
<!ATTLIST match
          address CDATA #REQUIRED
          port CDATA "*"
>
<!ELEMENT replace EMPTY>
<!ATTLIST replace
          address CDATA #REQUIRED
          port CDATA "*"
>
```

Chapter 14

Packet Loss

This section is about packet loss. Packet loss is most-often caused when some part of the system is receiving packets at a higher rate than it is able to process them. This typically results in queuing of incoming packets, but queues do not have unlimited size. If the incoming packets exceed the processing speed for too long a period of time, the queue will fill and packets will be dropped.

Packet loss is a fact of life in networks. Some users are able to provision and tune their systems such that they might only lose a few packets per week. Other users routinely live with several lost packets per minute. Many users do not monitor their system for loss and have no idea how frequent it is.

Packet loss is undesirable for many reasons. For reliable protocols (TCP, LBT-RM, etc), detection and retransmission of lost packets introduces significant latency. If packet rates are too high for too long a period of time, the reliability protocol can give up trying to recover the lost data. This can result in disconnects (for TCP) or delivery of "unrecoverable loss" events, where application messages can be lost forever.

14.1 UM Recovery of Lost Packets

UM recovers lost packets at multiple levels:

- **Transport** - TCP, LBT-RU, LBT-RM have low-level handshakes to detect and retransmit lost packets.
- **OTR/Late Join** - independent of transport, OTR and Late Join will recover data, typically after the transport layer gives up.
- **Persistence** - closely-associated with OTR and Late Join, the Persistent store provides a much greater capacity to recover lost data.

One fundamental problem with most UM use cases is that users want the flow of new messages to continue unimpeded in parallel with recovery efforts of lost packets. Given that packet loss is almost always a result of high packet rates overloading one or more queuing points along a messaging path, the addition of packet recovery efforts can make the overload even worse. "Pouring gasoline on a fire" is an often-repeated metaphor.

Fortunately, packet rate overload tends to be temporary, associated with short-term traffic bursts. That is one reason why the UM lost packet recovery algorithms use time delays. For example: **transport_lbtrm_nak_initial_backoff↩ _interval (receiver)** and **otr_request_initial_delay (receiver)**. By waiting before requesting retransmissions, the burst is allowed some time to subside before we add retransmission to the normal traffic load. These delays do add to latency, but shortening the delay too much risks making the loss worse, which can make the overall latency worse than having a longer delay.

One limiting factor related to data recovery is the amount of data which is buffered and is available for retransmission. Applications need to continue sending new data while recovery takes place. Since the buffer is of limited size, older messages will eventually be overwritten with new messages.

For streaming applications, these buffers are held in memory, and the sizes are usually measured in megabytes. For persistent applications, the Store writes its buffer to disk, allowing for buffer sizes orders of magnitude larger than memory-based buffers. But even the Store's disk-based buffer is of finite size, and is susceptible to being overwritten if it takes too long to recover data.

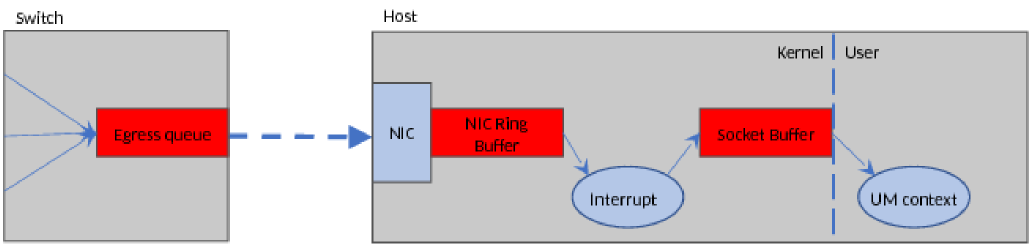
Note that the **Receiver Paced Persistence (RPP)** feature seeks to maximize the reliability of messaging by allowing the publisher to be blocked from sending rather than overwriting unacknowledged data.

Finally, it is very important for UM users to make use of UM's extensive monitoring capabilities. Since UM does a good job of recovering lost packets, you may be experiencing high latency spikes without knowing it. Also, if loss is bad enough, UM will be unable to fully recover all the lost data, and unrecoverable loss can result. User are strongly advised to monitor transport statistics and pay special attention to receivers that repeatedly experience loss. Even if that loss is successfully recovered, you should diagnose and treat the loss before it gets worse and becomes unrecoverable loss.

See **Monitoring Transport Statistics** for more information.

14.2 Packet Loss Points

There are just a few common points at which packets are normally lost:



The red buffers/queues are the most common locations where packets are typically lost during a packet burst.

14.2.1 Loss: Switch Egress Port

The switch egress port can come under pressure if data flows from multiple sources need to be merged onto a single outgoing link. The outgoing link can be an internal trunk or communication link connecting two pieces of network equipment, but more typically it is link to a destination host.

It is easy to understand how loss can happen here. Suppose three remote hosts are sending UDP data streams at the destination host. If each stream is carrying 0.5 gigabit/sec of throughput, the switch needs to send 1.5 gigabit/sec over a 1 gigabit link, a clear overload. If this is a very short-term burst, the egress queue will hold the data until the incoming data flows subside and the outgoing port can get caught up. But if the burst lasts too long and the egress queue fills, the switch has no choice but to drop packets.

Note that the switch will not count these drops as "errors". There is a separate drop counter which should be examined to diagnose switch egress port loss.

MITIGATION

The only solution is to reduce the packet rate being sent to the destination host. Due to the way publishers map topics to Transport Sessions, it is often the case that the receiver will be discarding messages that it hasn't subscribed to. For the LBT-RU transport type, this can often be accomplished by turning on **Source Side Filtering**. With Multicast, the Source Side Filtering feature is not possible. So it is sometimes necessary to change the topic mapping, usually by increasing the number of multicast groups, and thus reducing the number of topics per session.

By grouping topics with knowledge of receiver interest, you can reduce the number of unsubscribed topics being received.

14.2.2 Loss: NIC Ring Buffer

As packets are received by the host's NIC (Network Interface Card), they are copied into host memory in a structure called the Receive Ring Buffer. The NIC interrupts the OS, which has the responsibility to unload the packet buffers from the Ring Buffer. If the incoming packet rate is faster than the OS can unload the Ring Buffer, it will fill and packets will be dropped.

Normally, the kernel is able to service NIC interrupts without any trouble. However, there is one situation which can put the Ring Buffer under pressure: When multiple processes on the host are subscribed to the same multicast stream, the kernel must replicate and deliver the packets to each process. For a small number of processes (5-10), the kernel will still be able to keep up with the incoming packets.

However, as companies consolidation servers by moving to large, many-core hosts (often virtualized), we see the same multicast stream subscribed to by increasing numbers of processes on the same physical server. We have seen NIC Ring Buffer loss (also called "overrun") with as few as 15 processes subscribed to a heavy stream of multicast packets.

(Note that this is generally only a problem for multicast. With Unicast data distribution to many recipients, the source essentially does the packet replication work. The receive-side work for the kernel for each unicast packet is minimal.)

MITIGATION

Users should maximize the size of the NIC's Receive Ring Buffer. For many NICs, the size of the ring buffer is configured by the number of receive descriptors. This should be set to the maximum allowable value.

The mitigators listed above will also help this problem by reducing the incoming packet rate.

Another solution is to spread processes across more physical hosts. This has the additional advantage of reducing latency, since multicast replication within a host must be done in software by the kernel and is serial in nature, whereas replication in the network is done by specialized hardware in parallel.

Another possible solution involves the use of the DRO as the primary receiver of the multicast data, which then re-publishes it on the host using the IPC transport. This has the disadvantage of introducing some additional latency since the messages must be received by the DRO and then forwarded to the application receivers. It also requires separating the applications to their own Topic Resolution Domains (TRDs).

14.2.3 Loss: Socket Buffer

The Socket Buffer represents the interface between the OS kernel and the user process. Received data is transferred from the NIC Ring Buffer to the destination Socket Buffer(s). The Socket Buffers are then emptied by the application process (in this case, the UM Context Thread). Socket buffer sizes are configurable, according to the transport type. For example, see **transport_lbtrm_receiver_socket_buffer (context)**.

The TCP protocol is designed to ensure that the socket buffer cannot be overflowed. However, UDP-based protocols (LBT-RU and LBT-RM) are susceptible to socket buffer overflow, which leads to datagram loss.

MITIGATION

All of the mitigators listed above will help this problem by reducing the incoming packet rate.

In addition, the process of unloading the socket buffer can be made much more efficient by configuring **multiple_receive_maximum_datagrams (context)** to a reasonably small value, typically 10.

Also, the Transport Services Provider (XSP) feature can help by splitting the work of unloading multiple sockets across multiple threads.

14.2.4 Loss: Other

The three loss locations described above are all related to high packet rates causing fixed-sized packet buffers to overflow. These represent by far the most common reasons for packet loss. However, it is possible that you will experience loss that cannot be diagnosed to those three causes.

For example, we have seen reports of NIC hardware malfunctioning such that most packets are successfully received and delivered, but some percentage of packets fail. At least one user reported that a misconfigured router "flapped" a route, resulting in periodic, short-term loss of connectivity between two sub-networks. We have seen a case where the use of kernel bypass drivers for high-performance NICs (specifically Solarflare) can cause multicast deafness if both accelerated and non-accelerated processes are run on the same host. We have even seen a case where replacing the Ethernet cable between a host and the switch resolved packet loss.

It is not possible to have a step-by-step diagnostic procedure which will pinpoint every possible cause of packet loss. The techniques described in this document should successfully diagnose a large majority of packet loss causes, but nothing can replace your infrastructure network engineers expertise at tracking down problems.

14.3 Verifying Loss Detection Tools

The preceding techniques for mitigating loss are best deployed after you have identified the type of loss. Unfortunately, we have found that the tools available to detect and identify the location of loss to be problematic. Informatica does not provide such tools, and does not follow the market for such tools to find a reliable supplier.

However, we have a starting point that has given us some measure of success in diagnosing the loss points. It is important that you try out these tools to verify that they properly detect the different types of loss. In order to verify them, you need to be able to reproduce on demand loss at each of the points: switch, NIC, and socket buffer.

Fortunately, this is reasonably easy using the `msend` and `mdump` tools provided in the "mtools" package offered by Informatica free of charge. Download the mtools package from https://community.informatica.com/solutions/informatica_mtools The source files for `msend` and `mdump` are provided, as well as pre-built binaries for most major platforms.

Informatica recommends verifying your loss diagnosis tools *before* you have a serious loss event that disrupts your application system, preferably before your system goes into full production usage. Periodically running and recording the results of these tools during normal operation will make it possible to diagnose loss after the fact. Detecting and identifying non-severe (recoverable) loss can be used to prevent serious (unrecoverable) loss events in the future.

14.3.1 Prepare to Verify

1. Download and install mtools on two hosts, designated "sender" and "receiver". Informatica recommends that the hosts be "bare metal" (not virtual machines), and that they be connected to the same switch. This minimizes the chances that the verification tests will cause any disruption to normal operation.

2. Contact your system and network administrators and set up some time that they can work with your during the verification process. They will need to perform operations that you probably do not have the ability to do.

3. Have the network administrator allocate a multicast group that you can use for this test. That multicast group should be otherwise unused in your organization. Warn the administrator that you will be pushing intense

traffic bursts between the two hosts.

14.3.2 Verifying Switch Loss

A possible Unix command that a network administrator could use is:

`snmpwalk -v 1 -c public` *SWITCH_ADDR* `IF-MIB::ifOutDiscards`

Note that the above community string ("public") is probably not enabled; the network administrator will know the appropriate value. Ideally, the network administrator would run that command every 5 or 10 minutes, logging to a file, with a time stamp. If this log file could be shared read-only to the project groups, they can time-correlate any unusual application event with loss reported by the switch.

To verify that you properly detect switch loss, follow these steps:

1. Work with your system and network administrators to **enable** Ethernet flow control in both the switch port and the NIC.
2. Use the above `snmpwalk` command (or equivalent) to record the current drop counts for the switch ports.
3. On the receiving host, run 30 copies of the following command:
   `mdump -q` *MCAST_ADDR* `12000` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the receiving host.
4. On the sending host, run the following command:
   `msend -5` *MCAST_ADDR* `12000 15` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the sending host.
5. When the test completes, use the `snmpwalk` command again (or equivalent) to record another set of drop counters. The receiving host's drop count should be larger.

This test works by making the receiving host's kernel work very hard for each received datagram. It should be unable to keep up. (If you don't see any drops caused by the test, try doubling the number of copies of `mdump` on the receiving host.) The Ethernet flow control settings on the NIC and switch will prevent NIC loss in its ring buffer by slowing down the switch's egress port. Thus, the switch's egress queue will fill and should overflow.

14.3.3 Verifying NIC Loss

Unix

On some Unix systems, the "ifconfig" command will accurately report receive overrun on the NIC. For example:
`ifconfig eth0`

But in many Unix systems, the values reported by "ifconfig" remain at zero, even when the NIC has in fact overrun its receive ring buffer. We recommend also trying the "ethtool" command. For example:
`ethtool -s eth0`

Windows

To the best of our knowledge, there is no standard Windows tool for detecting NIC loss. Some drivers might provide that information from the interface control panel. Otherwise, you might need to download a management application from the NIC or system vendor.

If you know of a widely-available method to detect NIC overrun on Windows, please let us know at our **D↩ LMessagingBuilds** email account on informatica.com (that awkward wording used to avoid spam address harvesters).

To verify that you properly detect NIC loss, follow these steps:

1. Work with your system and network administrators to **disable** Ethernet flow control in both the switch port and the NIC.
2. Use your NIC loss tool to get the current receive overrun count.
3. On the receiving host, run 30 copies of the following command:
   `mdump -q` *MCAST_ADDR* `12000` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the receiving host.
4. On the sending host, run the following command:
   `msend -5` *MCAST_ADDR* `12000 15` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the sending host.
5. When the test completes, use the NIC loss tool again to record the receive overrun count.

This test works by making the receiving host's kernel work very hard for each received datagram. It should be unable to keep up. (If you don't see any drops caused by the test, try doubling the number of copies of `mdump` on the receiving host.) The lack of Ethernet flow control means that the switch will send the packets at full line rate, which should overflow the NIC ring buffer.

14.3.4 Verifying Socket Buffer Loss

On most systems, the `netstat` command can be used to detect socket buffer overflow. For example:
`netstat -s`
Look in the UDP section for "receive errors". This normally represents the number of datagrams dropped due to the receive socket buffer being full.

Note that Windows prior to version 7 does not increment that field for socket buffer overflows. If you have pre-↩ Windows 7, we don't know of any command to detect socket buffer overflow.

To verify that you properly detect socket buffer overflow, follow these steps:

1. Use `netstat -s` to get the current receive error count.
2. On the receiving host, run a single copy of the command:
   `mdump -q -p1000/5` *MCAST_ADDR* `12000` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the receiving host.
3. On the sending host, run the following command:
   `msend -5 -s2200` *MCAST_ADDR* `12000 15` *INTFC_ADDR*
   where *MCAST_ADDR* is the multicast group for the test, and *INTFC_ADDR* is the IP address of the sending host.
4. When the test completes, use `netstat -s` again to get the new receive error count.

This test works by introducing a short sleep in the "mdump" command between reception of datagrams. This causes the socket buffer to overflow.

Chapter 15

UM Glossary

15.1 Glossary A

ABI - Application Binary Interface

The execution-time interfaces presented by one software system, generally in the form of a dynamic (shared) library, for use by other software systems. ABIs are generally considered to be in the realm of binary, compiled code, not source code. Two versions are considered ABI compatible if the dynamic libraries can be used interchangeably by an application without the need to rebuild or relink that application. See also API.

ACK - Acknowledge

Generally, a control message which acknowledges some event or condition. Within the context of Ultra Messaging, it is often used to refer to a persistence control message sent by a subscriber to the Persistent Store to indicate that it has completed processing of a given data message. See Persistence.

ACE - Access Control Entry

A filter specifier to control which topics are allowed to transit a UM Router portal. One or more ACEs make up an Access Control List (ACL). See **Access Control Lists (ACL)**.

ACL - Access Control List

A method used by the Dynamic Routing Option (DRO) to control which topics are allowed to transit a UM Router portal. An ACL consists of one or more Access Control Entries (ACE). See **Access Control Lists (ACL)**.

ActiveMQ

The name of an open-source JMS-oriented messaging system. The Ultra Messaging UMQ product contains an enhanced form of ActiveMQ to provide queuing semantics and a JMS API. See **UMQ Overview**.

AMQP - Advanced Message Queuing Protocol

An open standard messaging wire protocol. See `Wikipedia's write-up` for more information on AMQP. The UMQ product grouping makes use of AMQP to provide interoperability between Ultra Messaging and ActiveMQ. See **UMQ Overview**.

API - Application Programming Interface

The callable functions, classes, methods, data formats, and structures presented by one software system for use by other software systems. APIs are generally considered to be in the realm of source code, not compiled binaries. APIs are generally documented, and can be extended from one version to the next. Two versions are considered API compatible if the application can be built against either version interchangeably without the need to modify the source code. Ultra Messaging has APIs available for the C, Java, and .NET (C#) programming languages. For example, **lbm_context_create()** is part of the C API. See also ABI.

15.2 Glossary B

BOS - Beginning Of Stream

An event delivered to a receiver callback indicating that the link between the source and the receiver is now active. Be aware that in a deployment that includes the UM Router, it may only indicate an active link between the receiver and the local router portal, not necessarily full end-to-end connectivity. See also EOS

Broker

A daemon which mediates the exchange of messages. In the context of Ultra Messaging, it refers to the ActiveMQ daemon which implements the queuing functionality and JMS. See Queuing.

15.3 Glossary C

CIDR - Classless Inter-Domain Routing

Generally, CIDR refers to the division of a 32-bit IPv4 address between network and host parts. In the context of Ultra Messaging, CIDR notation can be used to ease the specification of host network interfaces. See **Specifying Interfaces**.

Context

Within the context of Ultra Messaging, a context is an object which functions conceptually as an environment in which UM runs. Context is often abbreviated as "ctx". See Context Object.

CTX - Context

Within the context of Ultra Messaging, a context is an object which functions conceptually as an environment in which UM runs. See Context Object.

15.4 Glossary D

DBL - Datagram Bypass Layer

A kernel-bypass driver that accelerates sending and receiving UDP traffic and operates with Myricom 10-Gigabit Ethernet adapter cards for Linux and Windows. See **Myricom® Datagram Bypass Layer (DBL™)**.

Delivery Confirmation

An optional event generated by a persistent subscriber's receiver and delivered to a persistent publisher's source to indicate that the subscriber has completed processing of a message. See Persistence.

Delivery Controller

An instance of the receive-side "topic layer" within the UM software stack. See UM Software Stack.

DLQ - Dead Letter Queue

With queuing, the Dead Letter Queue (DLQ) is a destination for messages that cannot be delivered to a receiver. See **Dead Letter Queue**.

DRO - Dynamic Routing Option

The name of an Ultra Messaging option which provides routing of messages different Topic Resolution Domains (TRDs). See Dynamic Routing Option (DRO).

Dynamic Routing Option (DRO)

The name of an Ultra Messaging option which provides routing of messages different Topic Resolution Domains (TRDs). The option consists of a daemon called the "\ref umrouter", or just the DRO. The UM Router is frequently used to span the bandwidth-limited links in a wide-area network due to the fact that it only passes messages for topics that are of interest. See UM Router.

15.5 Glossary E

EOS - End Of Stream

An event delivered to a receiver callback indicating that the link between the source and the receiver is deleted. Be aware that in a deployment that includes the UM Router, it may only indicate a deleted link between the receiver and the local router portal, not necessarily a full end-to-end link. See also BOS

Event Queue

Within the context of Ultra Messaging, an event queue object is a serialization queue structure and execution thread for delivery of other objects' events. Event queue is often abbreviated as "evq". See Event Queue Object.

EVQ - Event Queue

Within the context of Ultra Messaging, an event queue object is a serialization queue structure and execution thread for delivery of other objects' events. See Event Queue Object.

15.6 Glossary F

Flight Size

The number of messages that a persistent publisher can have outstanding that are not stable. A persistent publisher generally limits the number of unstable messages it can have outstanding, and may block further attempts to send until some outstanding messages become stable. See Persistence. See also Stability.

15.7 Glossary G

Gateway

An early version of a message router, replaced as of version 6.10 with the Dynamic Routing Option (DRO). See Dynamic Routing Option (DRO).

15.8 Glossary H

HF - Hot Failover

A form of redundancy in which multiple instances of a publisher send the same messages at the same time to subscribers, which select for application delivery the first copy received. If one publisher instance fails, the subscribers are able to continue operation receiving from the remaining publisher. See Hot Failover (HF).

HFX - Hot Failover eXtended

An extended form of redundancy in which multiple instances of a publisher send the same messages at the same time to subscribers, which select for application delivery the first copy received. If one publisher instance fails, the subscribers are able to continue operation receiving from the remaining publisher. HFX extends HF by allowing the subscribers to maintain the receiver objects in separate contexts, which gives greater flexibility in having the traffic use different network paths. See Hot Failover Across Multiple Contexts.

HRT - High Resolution Timestamp

A feature that leverages the hardware timestamping function of certain network interface cards to measure sub-microsecond times that packets are transmitted and received. See High-resolution Timestamps.

15.9 Glossary I

IPC - InterProcess Communication

Generally, the term simply refers to any of several mechanisms by which an operating system allows processes to communicate or share data. Within the context of Ultra Messaging, LBT-IPC specifically refers to the shared memory transport type. A source configured for LBT-IPC can only pass messages to receivers running on the same machine (or virtual machine). See Transport LBT-IPC.

15.10 Glossary J

JMS - Java Message Service

A standardized API for Java applications to send and receive messages. Ultra Messaging's UMQ product allows limited interoperability between applications using UM and applications using JMS. See **JMS**.

JNI - Java Native Interface

A method by which Java code can invoke code written in C.

15.11 Glossary L

LBM - Latency Busters Messaging

An old name of the Ultra Messaging product line. Superseded by UM. "LBM" is sometimes used to refer to the streaming product grouping. That use is superseded by "UMS". The abbreviation "lbm" lives on in various parts of the UM API, and was kept for backwards compatibility.

LBT - Latency Busters Transport

Usually used as a prefix for a specific transport type: LBT-RM, LBT-RU, LBT-IPC, and LBT-SMX. See **transport (source)**.

LJ - Late Join

A function by which a subscriber can create a receiver for a topic, and is able to retrieve one or more messages sent to that topic prior to the receiver being created. See Late Join.

LJIR - Late Join Information Request

A type of control message sent by a receiver to a source to request an Late Join Information control message. See Late Join.

15.12 Glossary K

Kernel-Bypass Driver

A device driver software package, normally supplied by a hardware vendor, which provides a user-space library and API for accessing the hardware without transitioning into the kernel. Some examples: Solarflare's Onload driver, Myricom's DBL driver, Voltaire's VMA driver.

15.13 Glossary M

MIM - Multicast Immediate Message

Alternate send method which makes use of a pre-configured LBT-RM transport which is shared by all like-configured applications. The "immediate" means that messages may be sent to arbitrary topic names without the creation of source objects. See Multicast Immediate Messaging. See also glossaryuim.

15.14 Glossary N

NAK - Negative AcKnowledgement

A type of control message sent by a receiver using LBT-RM or LBT-RU transports. Sent when packet loss causes a sequence number gap in received messages, the NAKs specify which sequence numbers are missing and request retransmission. See **Transport LBT-RM Reliability Options**.

NCF - NAK ConFirmation

A type of control message sent by a source using LBT-RM transport. The LBT-RM protocol requires a source send an NCF if it receives a NAK for which it is not willing to send a re-transmission. See **LBT-RM Source Ignoring NAKs for Efficiency**.

NIC - Network Interface Card

A part of a computer which connects to one or more network cables and provides packet-level communication to the operating system.

15.15 Glossary O

Open Onload

A kernel-bypass driver that accelerates sending and receiving UDP traffic and operates with Solarflare 10-Gigabit Ethernet adapter cards for Linux. See **Solarflare® Onload**.

OpenSSL - Open Secure Sockets Layer

A library which provides encryption services. OpenSSL is used by Ultra Messaging's encryption feature. See `https://www.openssl.org` for general information about OpenSSL. See Encrypted TCP for information about Ultra Messaging's encryption feature.

OTID - Originating Transport IDentifier

Control information which uniquely identifies a source object within a UM network. See **More About Proxy Sources and Receivers**.

OTR - Off-Transport Recovery

A method by which are lost and are not recoverable by the source transport can be recovered by UIMs using a method similar to Late Join. See Off-Transport Recovery (OTR).

15.16 Glossary P

PCRE - Perl Compatible Regular Expressions

An open-source library which closely implements the Perl 5 regular expression language. UM uses PCRE for wildcard receiver pattern matching. See `Wikipedia's write-up` for information on PCRE. See also UM Wildcard Receivers.

PDM - Pre-Defined Messages

A message encoding scheme based on integer field identifiers for structured messages can be assembled and sent by applications. Includes field types and performs data marshaling across different CPU architectures. See Pre-Defined Messages. See also SDM.

Persistence

A form of messaging, sometimes called "guaranteed messaging", in which messages sent by a publisher are temporarily saved in non-volatile storage so that subscribers can recover missed messages under a variety of failure scenarios. See Persistence.

PGM - Pragmatic General Multicast

A standards-based protocol for reliable multicast. Ultra Messaging's "LBT-RM" protocol is inspired by PGM. See Transport LBT-RM for a list of differences between LBT-RM and PGM..

Portal

An interface to the UM Router. A UM Router portal can either be an endpoint portal (interfaces with a Topic Resolution Domain), or a peer portal (interfaces with another UM Router). See **UM Router Portals**.

PTP - Precision Time Protocol

A protocol used to synchronize clocks throughout a computer network. Used by some NICs to synchronize host clocks (e.g. Solarflare). See `Wikipedia's write-up` for more information.

Pub/Sub - Publish / Subscribe

A model of messaging passing in which the publisher (sender) does not keep track of the subscribers (intended recipients) of messages. Instead, the messages carry metadata (topic name) in which the subscribers express interest, and the underlying messaging software forwards messages to the subscribers based on that interest.

15.17 Glossary R

RCV - Receiver

Within the context of Ultra Messaging, a receiver is an object used to subscribe to a topic. "Receiver" is sometimes used to refer generally to an entire subscribing application. See Receiver Object. See also Wildcard Receiver.

Receiver

Within the context of Ultra Messaging, a receiver is an object used to subscribe to a topic. "Receiver" is sometimes used to refer generally to an entire subscribing application. Receiver is often abbreviated as "rcv". See Receiver Object. See also Wildcard Receiver.

Registration

When a publisher creates a persistent source, that source must register with the configured Persistent Stores before it can start sending messages. This registration prepares the Persistent Store and the source to co-operate in the transfer of persisted messages. Likewise, when a subscriber creates a persistent receiver, that receiver must register with the configured Persistent Stores before it can start receiving messages. See Persistence.

RM - Reliable Multicast

A shortening of "LBT-RM". The Ultra Messaging protocol and implementation in which user messages sent via Multicast UDP are monitored for loss, and retransmissions are arranged to recover loss. See Transport LBT-RM.

RPP - Receiver-Paced Persistence.

A form of persistence in which a publisher can be blocked from sending if receivers are having trouble keeping up with the message rate. See Persistence. See also SPP.

Router

Within the context of Ultra Messaging, "\ref umrouter" generally refers to the daemon within the Dynamic Routing Option (DRO). See Dynamic Routing Option (DRO).

RSA - Rivest, Shamir, and Aleman

A public-key cryptosystem developed by Ron Rivest, Adi Shamir, and Leonard Adleman. Included in the Open↔SSL library used by Ultra Messaging's encryption feature. See Encrypted TCP.

RU - Reliable Unicast

A shortening of "LBT-RU". The Ultra Messaging protocol and implementation in which user messages sent via Unicast (point-to-point) UDP are monitored for loss, and retransmissions are arranged to recover loss. See Transport LBT-RU.

RX - Re-transmission

Depending on the context, RX can either mean the messages retransmitted by the LBT-RM and LBT-RU transport protocols (e.g. in transport statistics), or it can mean the messages recovered via the Persistent Store or Late Join.

15.18 Glossary S

SDM - Self-Describing Messages

A message encoding scheme based on keyword-value pairs for structured messages can be assembled and sent by applications. Includes field types and performs data marshaling across different CPU architectures. See Self Describing Messaging. See also PDM.

SM - Session Message

A type of control message used by the LBT-RM protocol to keep a Transport Session alive. See **Transport LBT-RM Operation Options**.

SNMP - Simple Network Management Protocol

A standardized protocol by which computers and network equipment can be monitored and managed from a central point (management station). SNMP is also the name of an Ultra Messaging option which makes UM application usage statistics available for monitoring by a standard SNMP management station.

Source

Within the context of Ultra Messaging, a source is an object used to send messages to a topic. "Source" is sometimes used to refer generally to an entire publishing application. Source is often abbreviated as "src". See Source Object.

SPP - Source-Paced Persistence.

A form of persistence in which a publisher is allowed to continue sending at its natural rate, even if one or more receivers are falling behind to the point that the message repository's oldest messages are overwritten, leading to unrecoverable loss. See Persistence.

SRC - Source

Within the context of Ultra Messaging, a source is an object used to send messages to a topic. "Source" is sometimes used to refer generally to an entire publishing application. See Source Object.

SRI - Source Registration Information

A type of control message used to communicate persistence information between persistent publishers and subscribers. A subscriber of persistent messages needs an SRI to successfully register with a persistent store. See Persistence.

Stability

The state that a persistent publisher's sent message has been successfully persisted in the Persistent Store. In the time between message transmission and message stability, the message is at risk of being lost. The term is also used to refer to the source event delivered to a publishing application to indicate a message's stability. See Persistence. See also Flight Size.

Store

A shortening of "Persistent Store". An Ultra Messaging component which works with persistent sources and receivers to record messages, and also deliver previously-recorded messages for recovery. The UMP and UMQ product groupings include the Persistent Store; the UMS product grouping does not. See Persistence.

15.19 Glossary T

TIR - Topic Information Record

A type of topic resolution control message used by a source to advertise its details. Subscribers use TIRs to discover and connect to sources of interest. See Topic Resolution Overview.

TQR - Topic Query Record

A type of topic resolution control message used by a receiver to discover sources of interest. Publishers use TQRs to trigger the sending of TIRs. See Topic Resolution Overview.

TR - Topic Resolution

The protocol used by Ultra Messaging components to exchange information about available topics and topic interest. See Topic Resolution Overview. See also TIR and TQR.

Transport Session

A specific run-time instance of a transport type to carry application messages. The Transport Session can be thought of as a communications channel. As a publishing application creates sources, it maps those sources onto Transport Sessions. A Transport Session is fairly resource-intensive, so it is frequently the case that many sources are mapped to each Transport Session.

TRD - Topic Resolution Domain

A group of Ultra Messaging applications and UM components which communicate with each other directly, not through a UM Router. Specifically, it refers to those applications and components which directly exchange Topic Resolution control messages. Applications in different TRDs are not able to communicate with each other unless one or more UM Routers are used to interconnect the TRDs. See Topic Resolution Domain.

TSNI - Topic Sequence Number Information

A type of control message sent by a source to assist in the detection and recovery of certain types loss. See Loss Detection Using TSNIs.

15.20 Glossary U

UIM - Unicast Immediate Message

Alternate send method which makes use of pre-configured TCP transports. The "immediate" means that messages may be sent to arbitrary topic names without the creation of source objects. Sending a UIM bypasses Topic Resolution, so the calling application must specify the address information for the intended recipient. Because of this, the UIM feature is rarely used directly by user applications. However, Ultra Messaging uses UIMs internally for many of its control messages. See Multicast Immediate Messaging. See also glossarymim.

ULB - Ultra Load Balance

A feature of the Ultra Messaging UMQ product grouping which provides a limited subset of queuing semantics without the use of a central message broker. ULB is generally used to provide high-speed load balancing of UM messages. In the Pub/Sub model, if multiple subscribers create receivers for the same topic, each subscriber will receive a copy of every message sent. In the Queuing model, the messages are *distributed* to the multiple subscribers, with each message only being acted on by one of those subscribers. See **Ultra Load Balancing (ULB)**.

UM - Ultra Messaging

The name of the Informatica messaging middleware product line. UM is based on the pub/sub model of message passing, which allows the components of distributed applications to communicate. Note that Ultra Messaging is registered trademark of Informatica, LLC.

UM Router

Within the context of Ultra Messaging, "\ref umrouter" generally refers to the daemon within the Dynamic Routing Option (DRO). See Dynamic Routing Option (DRO).

UMCache - Ultra Messaging Cache

The name of an Ultra Messaging option which provides a limited degree of message storage and retrieval.

UMDS - Ultra Messaging Desktop Services

The name of an Ultra Messaging option which consists of a server daemon and a set of client libraries which provides simplified access to an Ultra Messaging network.

UME - Ultra Messaging, Enterprise edition

An old name of the UMP product grouping. Superseded by UMP. The abbreviation "ume" lives on in various parts of the UM API, and was kept for backwards compatibility.

UMM - Ultra Messaging Manager

A component of UM which allows users to centrally edit, store, and distribute configuration information to distributed applications. See the `UM Manager Guide`.

UMP - Ultra Messaging, Persistence edition

An Ultra Messaging product grouping which supports message streaming and persistence. The term is sometimes used to refer specifically to the persistence function. See Persistence.

UMQ - Ultra Messaging, Queuing edition

An Ultra Messaging product grouping which supports message streaming, persistence, and queuing. The term is sometimes used to refer specifically to the queuing function. See Queuing.

UMS - Ultra Messaging, Streaming edition

An Ultra Messaging product grouping which supports message streaming. The term is sometimes used to refer specifically to the streaming function.

15.21 Glossary V

VMA - Voltaire Messaging Accelerator

A kernel-bypass driver that accelerates sending and receiving UDP traffic and operates with Mellanox 10-Gigabit Ethernet and Infiniband adapter cards for Linux. (The software used to be owned by a company called Voltaire, which was acquired by Mellanox.) See **UD Acceleration for Mellanox® Hardware Interfaces**.

15.22 Glossary W

Wildcard Receiver

An object created by an application using the UM API to subscribe to a group of topics based on a Regular Expression pattern match. See UM Wildcard Receivers. See also PCRE. See also Receiver.

15.23 Glossary X

XSP - Transport Services Provider

An object created by a subscribing application to control the threading of message reception. See Transport Services Provider Object.

15.24 Glossary Z

ZOD - Zero Object Delivery

Feature which allows a Java or .NET subscribers to have received messages delivered without per-message object creation. This is more efficient than creating objects with each received message, and also avoids garbage collection. See Zero Object Delivery.