Informatica Migration Guide for TIBCO® Rendezvous®

July 2010

Table of Contents

To assist **29West** (http://www.29West.Com/) customers and potential customers migrating to our high performance messaging products, we provide this guide to help you convert to Informatica messaging solutions Latency Busters® **Messaging** (**UMS**), **Ultra Messaging® for the Enterprise** (**UMP**), or **Ultra Messaging Queuing Edition** (**UMQ**) from TIBCO® **Rendezvous®**.

1. Introduction

This migration guide is intended for an audience of application developers and system architects. It targets both existing applications and new applications, and details design choices you should consider early in the migration process, even if you decide not to pursue them.

Some other Informatica resources intended to supplement this Migration Guide:

- Knowledge Base (https://communities.informatica.com/infakb/kbexternal/default.aspx/) - use your Informatica Knowledgebase login to access the KB
- Product Documentation - see your install directory (../../readme.html)
- On-site Systems Engineer - contact your Sales representative to request an on-site consultation
- Support - go to http://29west.com/support

2. Application Changes and Notes

The Informatica architecture differs from daemon-based solutions like Rendezvous in order to provide the highest possible performance. As such you might expect to use it differently.

Informatica offers a simple, straightforward API to ease new application development. For existing applications, in certain situations, you may sometimes have a choice between making few changes to your application with little or no performance gain, vs. making bigger changes with a larger performance gain.

To help you understand these issues, each migration topic listed below offers a discussion of the performance improvement or functional difference. Some topics include a table with implementation instructions and references for further information. Note that some items require no action because they are defaults.

The list of topics (we recommend that you read them in order):

- *Ultra Messaging: How It Differs from Rendezvous*
- *Abstraction Layer*
- *Long Lived Subjects*
- *Short Lived Subjects*
- *Request/Response*
- *Wildcard Topics*
- *Topics and Transport Sessions*
- *Multicast Across a WAN*
- *Failover*
- *Threading*
- *Persistent Messaging*
- *Queuing*
- *Miscellaneous*

2.1. Ultra Messaging: How It Differs from Rendezvous

Before reading further, there are some essential terminology differences between daemon-based designs like Rendezvous and the .nothing in the middle. design of Ultra Messaging that you should understand.

Rendezvous applications send from publishers to subscribers and include a subject within each message payload to describe the destination for the response. Daemons called RVDs route messages from publishers to subscribers using configuration files to define the routes. Data transport is via reliable multicast for network hops between daemons, and TCP between daemons and applications. Rendezvous requires additional routing daemons called RVRDs to deliver messages to subscribers across a WAN.

Ultra Messaging applications send from sources to receivers based on shared interest in a topic that is not embedded within the message. Topics are arbitrary strings that facilitate the joining of receivers to sources via a real-time multicast topic resolution process. Standard network hardware routes messages from sources to receivers using Informatica configuration options (and, optionally, multicast groups) to define the routes. Data transport is via UDP reliable multicast (LBT-RM), UDP reliable unicast (LBT-RU), TCP, or inter-process communication on a single host (LBT-IPC). Informatica sends messages unicast or multicast over a WAN with no additional hardware and provides the UM Gateway for circumstances where multicast traffic cannot cross LAN boundaries.

2.2. Abstraction Layer

One big advantage of a customer-written messaging abstraction layer is the ability to hide any details related to the underlying messaging API that do not concern the application itself.

For example, to closely mimic Rendezvous behavior by allowing an application to publish to a topic without an explicit Ultra Messaging source create API call, you could implement the following.

1. Write a "pre_allocate_topic()" function which returns a handle for later use. Inside "pre_allocate_topic()", create the Ultra Messaging source (if not already created) via the **lbm_src_create()** API call and save the **lbm_src_t** pointer returned from that call for later use. Then create the handle, and save it with the **lbm_src_t** pointer for later retrieval.
2. Write a .publish(). function which accepts the handle returned from the pre-allocate call. Inside .publish()., use the handle supplied on input to retrieve the **lbm_src_t** pointer, and pass that pointer on the **lbm_src_send()** API call.

Another method for migrating from Rendezvous to Ultra Messaging is to make Ultra Messaging features optional in your abstraction layer, by supplying defaults for them and setting them to .off. initially, to be turned on whenever it suits your needs. This provides advantages to both existing and new applications.

- Existing applications may continue unchanged for a time (or forever, if you prefer).
- Existing applications can implement Ultra Messaging features in stages, when it makes sense for the application.
- New applications gain Informatica advantages sooner by using these features from the beginning, or phasing them in more aggressively.

2.3. Long Lived Subjects

To gain the biggest payback in low latency and high efficiency, your message streams should normally use standard UM sources with Topic Resolution. These streams could have some or all of the following characteristics.

- exist for long periods of time
- send a high volume of messages
- require the lowest latency

Other streams may benefit more from the recommendations in the *Short Lived Subjects* section.

The most important concepts to understand up front with Long-Lived Subjects are *Topic Resolution* and *Head Loss*.

2.3.1. Topic Resolution

Topic Resolution is the process of dynamically discovering topics, and then joining together all sources and receivers for those topics. Sources send topic advertisements, and receivers send topic queries, to all members of a Topic Resolution multicast group. Sources can also respond to topic queries with topic advertisements. Topic advertisements contain transport session information to enable the receiver to join the transport session.

If your network does not allow multicast, you can use unicast topic resolution with **lbmrd**, the UM Resolver Daemon.

To illustrate how Topic Resolution works, see the following simple example, and assume all of the following.

- use of multicast for both Topic Resolution and data transport
- receivers sending queries but no sources advertising

- a single transport session with one source application and one receiver application
- configuration options **resolver_multicast_address** and **resolver_multicast_port** set to their default values of **224.9.10.11** and **12965** respectively

The Topic Resolution steps for this scenario are shown below.

1. Receiver application starts, calls **lbm_rcv_topic_lookup()** API to start sending topic queries on multicast group 224.9.10.11 (by default, querying receivers send one query every 100 ms)
2. Twenty seconds later, source application starts, calls **lbm_src_topic_alloc()** API to create the topic object and send a topic advertisement (in response to the topic query it received) on 224.9.10.11
3. Receiver contexts listening on 224.9.10.11 receive topic advertisements, and store them for later lookup in the context-wide resolver cache
4. Source starts sending messages
5. Receiver calls **lbm_rcv_create()** API to:
   - create the receiver object on the topic
   - find the topic in the resolver cache
   - join the topic's multicast group
6. After a short delay for routers and switches to add the receiver-side transport IP to the proper routing tables, receiver joins source data stream and begins receiving messages (by default, now that a source is discovered, receiver stops sending queries)
7. Receiver continues to send topic queries every 100 milliseconds (by default) for re-transmitted topics and new topics just starting up

This is just one example of multicast topic resolution. For more details on other scenarios, including sources sending advertisements instead of receivers sending queries, late starting receivers, late starting sources, and repair of network problems, see Topic Resolution Scenarios (../../doc/Design/archictecture.html#TOPIC-RESOLUTION-SCENARIOS) in the UM Concepts Guide.

The only transport session types that do not participate in Topic Resolution are MIM and UIM (Multicast and Unicast Immediate Messaging). These transports embed the topic string within the message, like Rendezvous.

Note that wildcard receivers do not send topic queries by default. This is because every wildcard topic must be queried at every interval, even those that are already connected to sources. You may configure wildcard receivers to query, but we don.t recommend it, because of potentially high resource consumption.

You may turn off either topic advertisements or queries with configuration options; see Resolver Operation Options (../../doc/Config/resolveroperationoptions.html) in the Configuration Guide. However, either advertisements or queries must be on for Topic Resolution to work. If Topic Resolution is not suitable for your application environment, you should use Multicast Immediate Messaging.

Topic Resolution is configurable to allow accommodating a large volume of topics or to address loss, especially loss caused by socket buffers that are too small. See table at the end of this section for more.

2.3.2. Head Loss

Because the Topic Resolution process can typically take a few milliseconds to complete, Ultra Messaging sources that begin sending immediately can cause Head Loss, or unrecoverable loss of the initial messages in the stream. Head Loss may occur with multicast Topic Resolution for the following reasons.

- Source sends messages before receiver has fully resolved the topic and fully joined the sender's transport session (see step 6 in the above scenario). This is the most common form of head loss.
- Source sends a sudden burst of messages on a multicast address that network hardware has not yet "plumbed," resulting in the first messages being dropped while the switches and routers propagate multicast routing tables. This loss can happen even if the receiver had time to fully resolve the topic and fully join the multicast group.

There are a variety of ways to address head loss. We recommend these in order of preference:

- Source Pre-Creation - This is simply the practice of creating sources well ahead of when they are needed, such as during an application initialization step. For new applications, this might be easier than for existing applications.
- Late Join - The source must be configured to use Late Join (for receivers, Late Join is on by default). After sending messages, the source retains them in a Late Join buffer, up to the size limit of the buffer. At the receiver, Late Join processing uses information in the topic advertisement to request missing messages (via TCP). The source send the requested messages from its Late Join retention buffer, if they are found there. Note that Late Join introduces some initial latency and requires additional configuration and memory usage on the sending side. Since this extra memory is on a per-topic basis, it might not scale well with many thousands of topics.

2.3.3. Suggested Action Items

This table shows some specific action items you can take, and specific configuration options you can set, to implement some of the ideas mentioned above.

To implement source pre-creation:	Create sources well ahead of when they are needed, i.e., during application setup
To implement Late Join:	Configure both source and receiver as appropriate: **source late_join 1** **receiver late_join 1** Configure source Late Join buffer size (initial) with (for example): **source retransmit_retention_size_threshold 20000000** Configure source Late Join buffer size (max) with (for example): **source retransmit_retention_size_limit 30000000**

To configure Topic Resolution for more traffic	Configure both source and receiver with (for example): **context resolver_maximum_advertisements 5000** **context resolver_active_source_interval 500** **context context resolver_maximum_queries 1000** If using multicast topic resolution: **context resolver_multicast_receiver_socket_buffer 2000000** If using unicast topic resolution: **context resolver_unicast_receiver_socket_buffer 2000000**
Also see:	See Topic Resolution (../../doc/Design/archictecture.html#TOPIC-RESOLUTION) in Concepts Guide Resolver Operation Options (../../doc/Config/resolveroperationoptions.html) and Multicast Resolver N Options (../../doc/Config/multicastresolvernetworkoptions.html) in Configuration Guide See Using Late Join (../../doc/Design/lbm-features.html#USING-LATE-JOIN) in Concepts Guide and Join Options (../../doc/Config/latejoinoptions.html) in Configuration Guide

2.4. Short Lived Subjects

Some topics might not be ideal for normal UM sources and Topic Resolution because they only live for a short time or require the ability to begin sending immediately.

Therefore, these subjects might be best implemented using either Late Join, Multicast Immediate Messaging (MIM), or Unicast Immediate Messaging (UIM).

Refer to the Late Join discussion in *Long Lived Subjects*.

Multicast Immediate Messaging (MIM) is an alternative to the normal source-based UM model, and offers advantages for short-lived topics and applications that must send the first message on a topic immediately. MIM embeds the topic within the message and bypasses topic resolution by sending messages to all receivers in a pre-configured multicast group, allowing your application design to stay closer to a Rendezvous-like model. This extra information in the message payload, and the extra processing to filter out unwanted messages, does cause MIM to consume more network bandwidth and more CPU utilization at both source and receiver.

Both MIM and UIM transports cause UM to create a temporary transport session and then delete it automatically after a period of inactivity (defined by **mim_src_deletion_timeout**). Topics are sent to a single multicast address, and all sending and receiving applications must use that same multicast address, therefore, all filtering happens in UM, leading to higher latency on a per-message basis, compared to a source-based sender (e.g., LBT-RM).

Using MIM involves both configuration options and APIs.

2.4.1. Suggested Action Items

This table shows some specific action items you can take, and specific configuration options you can set, to implement some of the ideas mentioned above.

To implement Late Join:	Configure both source and receiver as appropriate: **source late_join 1** **receiver late_join 1** Configure source Late Join buffer size (initial) with (for example): **source retransmit_retention_size_threshold 20000000** Configure source Late Join buffer size (max) with (for example): **source retransmit_retention_size_limit 30000000**
To implement MIM:	Set in source configuration file: **source transport lbt-rm** Implement MIM APIs such as: **lbm_multicast_immediate_message()**
Also see:	See Using Late Join (../../doc/Design/lbm-features.html#USING-LATE-JOIN) in Concepts Guide and Join Options (../../doc/Config/latejoinoptions.html) in Configuration Guide See Multicast Immediate Messaging (../../doc/Design/lbm-features.html#MULTICAST-IMMEDIATE-MESSAGING) in Concepts Guide an lbm_multicast_immediate_message() (../../doc/API/lbm_8h.html#24e5bff3a70e571bb12024af67b47cb API Guide and MIM Network Options (../../doc/Config/multicastimmediatemessagingnetworkoptions. Reliability Options (../../doc/Config/multicastimmediatemessagingreliabilityoptions.html), and Operati Options (../../doc/Config/multicastimmediatemessagingoperationoptions.html) in Configuration Guide

2.5. Request/Response

There are several ways to implement the Request/Response use case with Ultra Messaging, and the built-in feature as described in Request/Response Model in the Ultra Messaging Concepts Guide is only one such way. While that method does offer the best performance, it usually requires more application changes. If you desire a more Rendezvous-like model, and are willing to trade performance for ease in migration, please contact Support for more details.

The key Request/Response differences between Rendezvous and UM are shown below in functional groups.

Rendezvous	Ultra Messaging
Defines hard-coded messaging routes between senders and receivers in RVD configuration files.	Discovers messaging routes in real-time with Topic Resolution
Uses short-lived subjects for responses. Request application normally stops listening after it receives the response.	Optimized for both short-lived and long-lived topics. Common in a topic defines an implicit connection between sources and r that exists for the life of that topic. Responses do not use topic

Rendezvous	Ultra Messaging
Response information (address and subject) explicitly embedded within message body by request application, where it must be extracted by response application.	Return address implicitly encoded in the request message head known location and length. Requests and responses matched u internally by UM APIs.
Sends requests unicast or multicast depending on RVD configuration options.	Sends requests via any supported protocol. UIM and MIM are analogous to RV unicast or multicast.

2.6. Wildcard Topics

Rendezvous wildcards use a limited design that allows multiple levels in a subject name, delineated by a period (".").
You can match an entire level using an asterisk ("*"), or all levels from that point to the end of the subject name using a right arrow (">"). Rendezvous wildcard expressions are case sensitive.

See the following table of example Rendezvous patterns and topics with matching results.

RV wildcard pattern	Matching results with sample text strings
hey.*	Matches "hey.you" and "hey.kid" but not "HEY", "hey", "say.hey", or "hey.kid.look"
hey.>	Matches "hey.you" and "hey.kid" and "hey.kid.look" but not "HEY", "hey", or "say.hey"

Informatica wildcards implement a powerful and common type of regular expression, called PCRE (Perl-Compatible Regular Expression). This section provides some examples and sample code to get you started. For more details, see this PCRE tutorial (http://perldoc.perl.org/perlre.html) (or this one (http://www.regular-expressions.info/tutorialcnt.html)).

See the samples in the table below:

PCRE string	Matching results with sample text strings
hey\.	Matches "hey.you", "hey.kid", and "hey.kid.look", but not "hey", or "say.hey"
\.hey>	Matches "say.hey", but not "hey.you", "hey.kid", "hey", or "hey.kid.look"
hey\..*\.look	Matches "hey.kid.look" and "hey.you.there.look.out" but not "hey.look"

Wildcard topics with Topic Resolution can sometimes cause excessive network traffic and CPU consumption at the receiver. If you use wildcards, we recommend one of the following mitigation techniques:

- Ensure that wildcard Topic Resolution queries are turned off (.off. is the default) and set sources to always advertise
- Consider using hypertopics (../../doc/API/lbmht_8h-source.html), a basic hierarchical system where like topics can be grouped by some initial set of characters

2.7. Topics and Transport Sessions

Sources send messages to topics, but ultimately those messages must be sent out across the network to the proper receivers. The transport session is the name for the UM construct that provides that functionality layer.

A transport session is defined in different ways depending on the protocol.

For LBT-RM, a transport session is composed of three segments:

1. source context
2. multicast group destination
3. outgoing port

For LBT-RU and TCP, a transport session is composed of two segments:

1. source context
2. listening port for incoming connections (for LBT-RU, implicit connections)

The way that you map your topics to transports helps you to control performance and balance the CPU and network load in your applications.

Please note that when a receiver subscribes to any single topic on a given transport session, the receiver's context receives all topics on that transport session even though some of them must be filtered out by UM due to lack of receiver application interest. This is a key point that has performance and design implications.

2.7.1. Suggestions for Mapping Topics to Transports

To keep your design closer to Rendezvous, or to minimize initial complexity, you can assign all topics in a single context to a single transport session. This can even be a viable permanent design if your overall messaging load is not too heavy, or if all topics have roughly equal performance needs. To dedicate a group of topics to a transport session, use the same source context, multicast group destination, and outgoing port.

However, we recommend that you break out your topics to specific transport sessions, especially if your application mix has:

- Distinct groups of topics that tend to be delivered to the same applications
- Prioritized topics that require the lowest possible latency, or consume high bandwidth, etc.

For example, if some group of receivers R always subscribe to topics A and B, while another group of receivers Q never do, you may want to group topics A and B onto a single dedicated transport session for R.

Likewise, if topics X and Y are more latency-sensitive than other topics, you may want to group topics X and Y onto a single dedicated transport. Note that the number of receivers is not a factor in this decision.

Using multiple transport sessions offers additional benefits by providing more per-transport resources:

- Transmission windows - for more granular management of timers and events
- Socket buffers - may be tuned per transport for best efficiency and throughput
- Session message traffic - heartbeat messages sent by UM for less-active sources

2.7.2. Kernel Overhead with Multiple Receivers on a Single Host

One more point to remember: with two or more receiving applications on a single host receiving from the same multicast transport session, the kernel must duplicate every message for every receiver because of the nature of multicast groups. The more receivers you have on a host, and the higher the bandwidth for that transport, the more

likely this is to lead to kernel overload, ultimately leading to potential data loss. Consider dedicating more transport sessions to the host and re-mapping topics to transports accordingly, or moving a subset of the receiver applications to another host.

2.8. Multicast Across a WAN

With Ultra Messaging, you don't need a daemon to send multicast over a WAN, except in the case where the network does not allow it. In that case, you can use a UM Gateway to convert multicast to unicast UDP or TCP. For more, see the UM Gateway (../../doc/Gateway/) documentation.

2.9. Failover

Informatica implements hot-failover with the concept of multiple redundant sources and a single receiver. To use hot-failover, follow these steps.

1. Deploy multiple UM source applications via the **lbm_hf_src_*()** family of API calls
2. (optional) Run each source on a separate host for best resiliency
3. Ensure each source sends the exact same messages
4. Deploy a single hot-failover receiver application via the **lbm_hf_rcv_*()** family of API calls to filter out duplicate messages and deliver only the first copy of any given message to the application

2.10. Threading

Rendezvous and UM create and use threads in very different ways. Generally speaking, Informatica messaging applications should need fewer threads than Rendezvous applications, leading to less thread contention.

Informatica applications generally use two threads per context: the UM or application thread for processing messages, and the context thread (one per context) to handle many internal tasks, including:

- Topic resolution
- Rate controller
- Timer management
- Transport management (such as packet/datagram retransmissions)

Some receiver applications, however, may use just the context thread if they can meet the required conditions (see Event Queue Objects (../../doc/Design/lbm-objects.html#EVENT-QUEUE-OBJECT) discussion about context thread callbacks).

The operational_mode configuration option determines the method used to create and terminate the context thread:

- automatically, when using **operational_mode embedded**, a.k.a. "embedded mode"
- manually by the application, and "donated" to UM, when using **operational_mode sequential**, a.k.a. "sequential mode"

To leverage threading parallelism even more, you can use Event Queues. The Event Queue design reduces bottlenecks by moving some critical-path functions from the context thread to an Event Queue thread, allowing more design freedom, functionality, and scalability, especially in the following areas.

- more freedom in creating/deleting UM objects (some objects cannot be created in the context thread)
- more freedom in callbacks regarding blocking and processing time
- fewer timing concerns with callbacks, such as interrupting critical sections of code

See Event Queue Objects (../../doc/Design/lbm-objects.html#EVENT-QUEUE-OBJECT) in the Design Guide for more.

Informatica offers many ways to customize your Ultra Messaging applications to use threads for maximum efficiency and lowest latency:

- use Event Queues to manage UM events for you, and free your application thread from timing and API constraints, especially on multi-processor or multi-core machines
- use sequential mode and a "donated" context thread to allow your application to set the priority you need for the context thread
- set processor affinity for a receive context thread to the CPU core that services NIC interrupts to get the lowest latency (this may limit throughput, however)

Any latency outliers you may encounter might be due to thread scheduling. Also, note that a real-time operating system can make latency more deterministic, but not necessarily lower.

2.11. Persistent Messaging

Ultra Messaging Persistence Edition (UMP) is Informatica's persistence messaging enterprise platform. UMP leverages the common UM API to provide persistence to all UM sources and receivers, and uses a unique design called Parallel Persistence to allow the fastest possible persistence capability, even during failover processing. UMP is a superset of UMS.

To implement Parallel Persistence, configure at least one UMP store. For production, you should also consider your strategy for the following:

- Failover - either round-robin (one single store operating at a time) or quorum/consensus (multiple stores in use concurrently)
- Registration IDs - tags that uniquely identifies a source or receiver to a particular store, for persistence and recovery
- Delivery confirmations - the minimum number of unique confirmations from different receivers before a message is released, set by the source

All UM sources and receivers in the UMP topic name space automatically gain the advantage of persistence.

For more, please see the UMP Concepts (../../doc/UME/umeconcepts.html) section and other relevant sections of the UMP Guide.

Also, you can realize some level of load balancing by breaking out topics evenly across transport sessions. For more, see *Topics and Transport Sessions*.

2.12. Queuing

Ultra Messaging Queuing Edition (UMQ) is Informatica's queuing enterprise platform. Sources can submit messages asynchronously to a queue, while receivers can later retrieve them in a separate asynchronous manner. UMQ is a superset of UMP.

To implement queuing for a source, set source configuration option umq_queue_name. All other Queuing options are set in the UMQ configuration file. For production, you should consider your strategy for the following:

- Failover - quorum/consensus (multiple queues in use concurrently)
- Registration IDs - tags that uniquely identifies a source or receiver to a particular store, for persistence and recovery

For more, please see the Designing Queuing Applications (../../doc/UME/designing-queuing-applications.html) section and other relevant sections of the UMQ Guide.

2.13. Miscellaneous

2.13.1. Starter Configuration Files

Informatica provides a variety of starter configuration files, for example, see LowestLatency.cfg (../../doc/Config/Examples/LowestLatency.cfg). Other files are available in the Examples (../../doc/Config/Examples/) directory. For more, see Example Configuration Scenarios (../../doc/Config/examples.html) in the Configuration Guide.

2.13.2. Automatic Monitoring

This feature allows monitoring applications to easily gain access to various levels of statistics, from context to transport to source and receiver. To monitor all transports within a context, set configuration option **context monitor_interval**. To monitor all Event Queues in a context, set **event_queue monitor_interval**. See Automatic Monitoring Options (../../doc/Config/automaticmonitoringoptions.html) in the Configuration Guide for more.

2.13.3. Event Queue Growth

Use configuration option **event_queue queue_size_warning** to receive notifications when the event queue size is larger than a given threshold. See Event Queue Options (../../doc/Config/eventqueueoptions.html) in the Configuration Guide for more.

2.13.4. Self Describing Messaging

Self Describing Messaging (SDM) is useful for cross platform messaging, particularly endian-ness conflicts, because it codifies all data in character format with meta-data to describe each field. Also, SDM simplifies messaging application maintenance, since receiver-side code doesn.t have to change when the format of the message changes.

For best performance, we recommend that you turn off the name tree wherever possible (see API documentation for lbmsdm.h (../../doc/API/lbmsdm_8h.html#_details)). Please note that turning off the name tree requires the implementation of one or more of the following design approaches in your applications.

- Use an iterator - spin through all the fields of a received message sequentially using an iterator instead of accessing randomly by name
- Manage field names manually - ensure field name uniqueness in source application
- Use indexes - immediately after adding a field at the source, save the index value locally for re-use when building subsequent messages

Note that you must turn off the name tree explicitly in order to gain the maximum performance advantage.

For more, see Self Describing Messaging (../../doc/Design/lbm-features.html#SELF-DESCRIBING-MESSAGING) in the Concepts Guide.