A picture containing logo

Description automatically generated

**Ultra Messaging Diagnostics Recording Recommendations**

By Steve Ford – [sford@informatica.com](mailto:sford@informatica.com)

This document describes Informatica’s recommendations for recording of Ultra Messaging diagnostic information. The goal is to provide enough visibility to resolve UM connectivity problems, including message loss.

Here are the recommended changes, in order of importance:

1. Record all failed API calls with informational string.
2. Record all non-message events.
3. Record all asynchronous UM log messages (strings).
4. Record UM transport statistics.
5. Record Topic-oriented API calls (using the code I wrote for you).

Recording this information can consist of writing to a local log file or database or sending the information to a centralized monitoring tool. Informatica generally recommends using both a local log file and a centralized monitoring tool.

In the code examples shown below, local log messages are written to standard output as an illustration. These should be replaced with the appropriate application logging function. The code fragments can also be found in "TmonExample/TmonExample.cs", a fully-working example application (both publisher and subscriber).

**1 - Record all failed API calls with informational string**

Every call to a UM function can throw an exception. These exceptions should be caught and the information contained therein should be logged.

In almost all cases where UM throws an exception, it doesn't make much sense for an application to attempt to continue execution. So except for very specialize cases, we recommend UM's exceptions be treated as unhandled exceptions, and should be caught and logged at the top level of the application. For example, in TmonExample/TmonExample.cs:

static void Main(string[] args) {

TmonExample example = new TmonExample();

try {

example.Run(args);

} catch (Exception e) {

example.Log("Exception: " + e.Message);

Console.WriteLine(e);

}

} // Main

**2 - Record all non-message events**

Much of the UM API operates by asynchronous callback, such as a UM receiver object which calls your "OnReceive()" callback function. But reception of application messages is only one kind of event that UM delivers to your application via the OnReceive callback. Those other events should be recorded because they can give important information about the health of data connections.

A typical UM receiver callback is structured as follows:

public int OnReceive(object cbArg, LBMMessage msg) {

switch (msg.type()) {

case LBM.MSG\_DATA:

HandleReceivedData(msg);

break;

case LBM.MSG\_BOS:

Log("app\_rcv\_callback: LBM\_MSG\_BOS: ["

+ msg.topicName() + "][" + msg.source() + "]");

break;

case LBM.MSG\_EOS:

Log("app\_rcv\_callback: LBM\_MSG\_EOS: ["

+ msg.topicName() + "][" + msg.source() + "]");

\_messagesReceived++;

break;

case LBM.MSG\_UNRECOVERABLE\_LOSS:

Log("app\_rcv\_callback: LBM\_MSG\_UNRECOVERABLE\_LOSS: ["

+ msg.topicName() + "][" + msg.source() + "]");

\_messagesReceived++;

break;

case LBM.MSG\_UNRECOVERABLE\_LOSS\_BURST:

Log("app\_rcv\_callback: LBM\_MSG\_UNRECOVERABLE\_LOSS\_BURST: ["

+ msg.topicName() + "][" + msg.source() + "]");

\_messagesReceived++;

break;

default: // Unexpected receiver event.

Log("app\_rcv\_callback: default: ["

+ msg.topicName() + "][" + msg.source() + "], type=" + msg.type());

break;

} // switch

return 0;

} // OnReceive

Later in this document (section 5), additional event recording will be added to OnReceive().

**3 - Record all asynchronous UM log messages**

An instance of UM (an LBM context) contains a thread which runs asynchronously to the application threads. Sometimes that thread detects problems that needs to be passed to the application for recording. This is done via the UM log callback.

A typical UM log callback is structured as follows:

private void Logger(int logLevel, string message) {

switch (logLevel) {

case LBM.LOG\_ALERT: MyLog("Alert: " + message); break;

case LBM.LOG\_CRIT: MyLog("Critical: " + message); break;

case LBM.LOG\_DEBUG: MyLog("Debug: " + message); break;

case LBM.LOG\_EMERG: MyLog("Emergency: " + message); break;

case LBM.LOG\_ERR: MyLog("Error: " + message); break;

case LBM.LOG\_INFO: MyLog("Info: " + message); break;

case LBM.LOG\_NOTICE: MyLog("Note: " + message); break;

case LBM.LOG\_WARNING: MyLog("Warning: " + message); break;

default: MyLog("Unknown: " + message); break;

}

} // Logger

Then, in the applications initialization code:

LBM lbm = new LBM();

lbm.setLogger(this.Logger);

Note that this should only be done once, and the class hosting the logger callback should have a single instance which exists for the entire time UM is active.

**4 - Record UM transport statistics**

It is possible to write UM transport statistics to a local log file, but it is generally not worth the effort. Instead, Informatica recommends sending this data to a centralized monitoring tool.

This can be done without any changes to your application source code. The following three or two configuration items should be added to the UM configuration (the first one is optional).

For "flat" LBM configuration files, use:

context monitor\_appid tmon\_example1

context monitor\_interval 300

context monitor\_transport\_opts config=mon.cfg

For an XML LBM configuration file, use:

<options type="context">

<option name="monitor\_appid" default-value="example1"/>

  <option name="monitor\_interval" default-value="300"/>

  <option name="monitor\_transport\_opts" default-value="config=mon.cfg"/>

The 300 second (5 minute) monitoring interval controls how often the monitoring data is sent. For some applications, several tens of K worth of data might be sent in each cycle. For testing purposes, you might want to reduce the time to 5 seconds just to shorten the tests. For production, you might to with an even longer interval, depending on the amount of data generated.

You should record ALL fields and make them available for review. Most of the numbers are cumulative since the application initialized. The most interesting fields are listed below.

**4.1 - Source statistics:**

You'll get one record per transport session, not per topic. A transport session is identified by a "source string". For example:

Source: LBTRM:192.168.1.20:12090:a47b4ae0:239.101.3.11:12000

Using transport statistics alone, you cannot determine which topics are carried by which transport sessions. However, if you also look at recorded BOS events (from "2 - Record all non-message events"), you'll see topics that are on that transport session (but only the topics that the application has subscribed to).

If you take the number of transport sessions and divide it into the context statistic "Topics in source topic map", you'll get the average number of topics per transport session. Some users have a 1-to-1 mapping of topics to transport sessions. Other users have 1000-to-1 or more.

|  |  |
| --- | --- |
| LBT-RM datagrams sent | Gives quantitative measure of bursts of messages. |
| LBT-RM NAKs received | Receivers send NAKs when they experience packet loss. |
| LBT-RM retransmission datagrams sent | If significantly different from NAKs received, indicates trouble. See "NAKs shed". |
| LBT-RM NAKs shed | Indicates the retransmission rate limiter prevented retransmissions. |

Note that "LBT-RM NAK packets received" is not as interesting as "NAKs received".

**4.2 - Receiver statistics:**

You'll get one record per transport session that the context has joined, not one per topic. A transport session is identified by a "source string". For example:

Source: LBTRM:192.168.1.20:12090:a47b4ae0:239.101.3.11:12000

Using transport statistics alone, you cannot determine which topics are carried by which transport sessions. However, if you also look at recorded BOS events (from "2 - Record all non-message events"), you'll see topics that are on that transport session (but only the topics that the application has subscribed to).

Be aware that a single receiver for one topic might join many transport sessions from different publishing applications. If N publishers have a source for topics "X", then a subscriber for topic "X" will join the N transport sessions that carry topic "X" (one from each publisher).

|  |  |
| --- | --- |
| LBT-RM datagrams received | Gives quantitative measure of bursts of messages. |
| Lost LBT-RM datagrams detected | This number increasing but no unrecoverable loss is a warning. It already indicates big latencies, and can turn into unrecoverable in the future. |
| LBT-RM NAKs sent | This number can be less than the lost datagrams (if multiple receivers have the same loss) or greater than the lost datagrams (if it has to NAK multiple times to get a retransmission). |
| LBT-RM datagrams unrecoverable (window advance) | Window advance means the source no longer has the datagram to retransmit. Maybe increase the source's configured transport\_lbtrm\_transmission\_window\_limit. |
| LBT-RM datagrams unrecoverable (NAK generation expiration) | The receiver's configured transport\_lbtrm\_nak\_generation\_interval might be too small? Or so much loss that it just can't get it all retransmitted in time? |
| LBT-RM LBM messages received with uninteresting topic | If this number is high, the source has too many topics mapped to the same transport session. This can cause loss. |

Note that "LBT-RM NAK packets sent" is not as interesting as "NAKs sent".

**4.3 - Context statistics:**

|  |  |
| --- | --- |
| Topic resolution datagrams received | If this increases at a rate that is a significant fraction of the receiver's "LBT-RM datagrams received" increase rate, your Topic Resolution might be causing loss. |
| Topics in source topic map | Shows the number of individual sources (topics) in this application |
| Topics in receiver topic map | Shows the number of individual topics that exist in the entire network. |
| Unresolved topics in receiver topic map | Unresolved topics either represents a topic resolution problem or receivers for topics that have no sources (application problem?). |

**5 - Record Topic-oriented API calls**

This makes use of the "Tmon" package, available at: https://github.com/UltraMessaging/tmon

Note that calling these Tmon API functions sends messages to the centralized monitoring tool, much like the transport statistics. But a big difference is that the transport statistics are sent from a UM periodic timer, whereas the "Tmon" reports are sent (as UM messages) from the application as it creates, closes, and makes method calls to Tmon objects.

Each Tmon object is associated with a corresponding UM object. Some of the Tmon API operations are made \*before\* calling the corresponding UM API, and some are called \*after\*. Please pay careful attention to the documentation below to know which is which.

Also note that whereas UM's C# API follows Java's "camelCase" convention for capitalization of public names, Tmon follows Google's "PascalCase" convention.

In the code fragments below, bold font is added code, normal font corresponds to your existing code.

**5.1 - TmonContext**

For each UM context that your application creates (typically one), create a "TmonContext" right after:

LBMContext ctx = new LBMContext(ctxAttr);

**TmonContext tmonContext = new TmonContext(ctx);**

**tmonContext.InitSender();**

The "InitSender()" API reports the context creation to the centralized monitoring tool.

For each UM context that your application deletes (closes), close the corresponding TmonContext right before. Note that all TmonSource, TmonReceiver, and TmonConnection objects under that TmonContext must already have been closed.

**tmonContext.Close();**

ctx.close();

The "Close()" API reports the context deletion to the centralized monitoring tool.

**5.2 - TmonSource**

For each UM source your application creates, create a "TmonSource" right before. Be sure to use the same topic name as the UM source.

**TmonSource tmonSrc1 = tmonContext.SourceCreate("src1");**

LBMSource src1 = ctx.createSource(ctx.allocTopic("src1", srcAttr));

The "SourceCreate()" API reports the source creation to the centralized monitoring tool.

For each UM source your application deletes (closes), close the corresponding TmonSource right after.

src1.close();

**tmonSrc1.Close();**

The "close()" API reports the source deletion to the centralized monitoring tool.

**5.3 - TmonReceiver**

For each UM regular (not wildcard) receiver your application creates, create a "TmonReceiver" with type "ReceiverType.Regular" right before. Use the same topic name as the receiver. Set up "source notification function" callback functions for when a receiver's delivery controllers get created and deleted (as part of joining sources). If you are already using source notification function callbacks, see Appendix 1.

**TmonReceiver tmonRcv3 = tmonContext.ReceiverCreate(ReceiverType.Regular, "src3");**

rcvAttr = new LBMReceiverAttributes("29west\_tmon\_context", "src3");

**rcvAttr.setSourceNotificationCallbacks(this.onDeliveryControllerCreate,**

**this.onDeliveryControllerDelete, tmonRcv3);**

LBMReceiver rcv3 = new LBMReceiver(ctx, ctx.lookupTopic("src3", rcvAttr),

this.OnReceive, null, null);

For each UM wildcard receiver your application creates, create a "TmonReceiver" with type "ReceiverType.Wildcard" right before. Use the same topic pattern as the wildcard receiver. Set up "source notification function" callback functions for when a receiver's delivery controllers get created and deleted (as part of joining sources). If you are already using source notification function callbacks, see Appendix 1.

**TmonReceiver tmonWRcv = tmonContext.ReceiverCreate(ReceiverType.Wildcard, "^.\*2$");**

LBMReceiverAttributes rcvAttr = new LBMReceiverAttributes("29west\_tmon\_context", "wc2");

**rcvAttr.setSourceNotificationCallbacks(this.onDeliveryControllerCreate,**

**this.onDeliveryControllerDelete, tmonWRcv);**

LBMWildcardReceiver wrcv = new LBMWildcardReceiver(ctx, "^.\*2$", rcvAttr,

null, this.OnReceive, null);

The ReceiverCreate() API reports the receiver creation to the centralized monitoring tool.

Both of these code fragments reference two new methods in your code: "OnDeliveryControllerCreate()" and "OnDeliveryControllerDelete()". These are explained below.

For each UM receiver (regular or wildcard) your application deletes (closes), close the corresponding TmonReceiver right after.

rcv3.close();

**tmonRcv3.Close();**

The Close() API reports the receiver or wildcard receiver deletion to the centralized monitoring tool.

**5.4 - TmonConnection**

As the application receivers and wildcard receivers discover matching sources and join those sources' transport sessions, UM will invoke the user's "source notification function" callback functions as UM creates the delivery controllers. The delivery controller can be thought of as a topic-level "connection" between a publisher and a subscriber.

To monitor those "connections", create a "TmonConnection" for each delivery controller, from inside the source notification function callback "create" function, and return the connection object as the return value of the callback function. This makes it available to the receiver callback as the per-source client data ("msg.sourceClientObject()").

In the TmonReceiver section above, source notification function callback functions OnDeliveryControllerCreate() and OnDeliveryControllerDelete() were set up. When UM joins a source and creates a delivery controller, the OnDeliveryControllerCreate() function is called:

**public object OnDeliveryControllerCreate(string sourceName, object cbArg) {**

**TmonReceiver tmonRcv = (TmonReceiver) cbArg;**

**MyLog("dc\_create\_cb: source\_name='" + sourceName + "'");**

**return tmonRcv.ConnectionCreate(sourceName);**

**} // OnDeliveryControllerCreate**

The OnDeliveryControllerCreate() function returns a TmonConnection object back to UM. The UM delivery controller will pass this back to the application's receiver callback function via msg.sourceClientObject().

When the receiver detects an end to a transport, the delivery controller is deleted and the OnDeliveryControllerDelete() is called. This is where the TmonConnection object should be deleted:

**public int OnDeliveryControllerDelete(string sourceName,**

**object cbArg, object sourceCbArg) {**

**TmonConnection tmonConn = (TmonConnection)sourceCbArg;**

**MyLog("dc\_delete\_cb: source\_name='" + sourceName);**

**tmonConn.Close();**

**return 0;**

**} // OnDeliveryControllerDelete**

The OnDeliveryControllerCreate() and OnDeliveryControllerDelete() APIs report the delivery controller create and delete events to the centralized monitoring tool.

**5.5 - Connection Lifecycle Events**

The events related to a connection's life cycle are:

1. Delivery controller creation
2. BOS (Beginning Of Session)
3. Data / Loss
4. EOS (End Of Session)
5. Delivery controller deletion

Not all connections go through all steps. For example, you can go from step 1 (delivery controller creation) straight to steps 4 and 5 (EOS and delivery controller deletion). If this happens repeatedly, it indicates multicast deafness.

The earlier call to ConnectionCreate() reports delivery controller creation, and the call to Close() records delivery controller deletion. The BOS, Loss, and EOS events must be recorded from the UM receiver application callback. The earlier recommendation, "2 - Record all non-message events", writes messages to the local log file.

The Tmon API can be used to send these events to the centralized monitoring tool. In your UM receiver callback:

public int OnReceive(object cbArg, LBMMessage msg) {

**TmonConnection conn = (TmonConnection)msg.sourceClientObject();**

**if (conn != null) {**

**conn.ReceiverEvent(msg);**

**}**

The TmonConnection object is stored in the message's source-specific client data as a result of the "return" statement from "OnDeliveryControllerCreate()". The call to ReceiverEvent() reports the BOS, Loss, and EOS events to the centralized monitoring tool.

**6 - Centralized Monitoring Tool**

The design and implementation of the centralized monitoring tool is the user's responsibility. The UM package includes an example program, "lbmmon.c", which performs the basic functions of collecting the monitoring statistics and printing human-readable data to standard output. Informatica recommends taking this program, which is provided in source form, and integrating it with your existing monitoring infrastructure.

As part of the development of the Tmon package, an enhanced example program is included: "lbmtmon.c". This adds the reception and printing of Tmon event reports.

Unlike the original "lbmmon.c" program, the reception and parsing of the Tmon event reporting messages is included in the source code.

**6.1 - Example Output**

Tmon: context create

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:27:58 2021 and 159008 usec

app\_context\_name=tmon\_example\_ctx

This report indicates that the application created a TmonContext object. The app\_id and app\_context\_name are extracted from the application context and are optional. The IP address and Process ID (PID) can be used to uniquely identify the application instance.

The "tmon\_ctx" is a unique Tmon object identifier within the application. I.e. another application process will also have "tmon\_ctx=1", but no other Tmon object in this application will have an identifier of 1. Note that applications might create multiple contexts (although it is not common).

Tmon: source create

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:27:58 2021 and 166024 usec

tmon\_src=6

topic=src1

This report indicates that the application created a TmonSource object, with identifier 6. It's parent Tmon context is identifier 1.

Tmon: wildcard receiver create

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:27:58 2021 and 160073 usec

tmon\_rcv=2

pattern=^.\*2$

This report indicates that the application created a TmonReceiver object with type Wildcard, with identifier 2. It's parent Tmon context is identifier 1.

Tmon: receiver create

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:27:58 2021 and 162428 usec

tmon\_rcv=3

topic=src1

This report indicates that the application created a TmonReceiver object with type Regular, with identifier 3. It's parent Tmon context is identifier 1.

Tmon: connection create

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:27:58 2021 and 168051 usec

tmon\_conn=7

tmon\_rcv=3

src\_str=LBTRM:192.168.1.20:12090:db1fd69:239.101.3.11:12000[2531965971]

This report indicates that the application created a TmonConnection object, with identifier 7. It's parent TmonReceiver is identifier 3. Note that the parent TmonReceiver object might be a regular receiver or a wildcard receiver.

The "src\_str" is the "source string", which identifies a specific transport session from a publisher. Remember that a transport session might have more than one topic mapped to it by the publisher. In this case, the bracketed number at the end identifies the individual source by its "topic index". However, there is not a simple way to determine the topic name associated with a topic index. But it can be done by using the "BOS" report (see below).

Tmon: connection BOS

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:03 2021 and 173013 usec

tmon\_conn=7

topic=src1

This report indicates that connection 7 delivered a BOS (Beginning Of Session). The topic name associated with the BOS event is "src1". For regular receivers, this should always be equal to the topic name associated with the creation of the receiver object. However, with wildcard receivers, this indicates the actual source topic matched.

The source string contained in the "connection create" report can be combined with the topic in the "connection BOS" report to associate a topic name with a source string and topic index.

Tmon: message loss

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:25 2021 and 664668 usec

tmon\_conn=15

sqn=2

This report indicates that connection 15 delivered an unrecoverable loss event. The sequence number is indicated. Note that the Tmon package is written to limit the number of loss reports from a given connection to once per second. E.g. if three unrecoverable loss events are delivered to a connection within the same second, only one report will be sent to the centralized monitoring tool. However, the Tmon package keeps track of the actual number of loss events, and reports them when the connection is closed.

Tmon: message burst loss

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=84041

tmon\_ctx=1

time=Sun Jan 3 10:49:44 2021 and 978262 usec

tmon\_conn=15

sqn=12

burst\_size=4

This report indicates that connection 15 delivered an unrecoverable burst loss event. The sequence number is indicated, as is the size of the burst. Note that the Tmon package is written to limit the number of loss reports from a given connection to once per second. E.g. if three unrecoverable loss events are delivered to a connection within the same second, only one report will be sent to the centralized monitoring tool. However, the Tmon package keeps track of the actual number of loss events, and reports them when the connection is closed.

Tmon: connection EOS

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:10 2021 and 418601 usec

tmon\_conn=7

msg\_count=1

loss\_count=0

burst\_count=0

dgrams\_lost=0

topic=src1

This report indicates that connection 7 delivered a EOS (End Of Session). The topic name associated with the EOS event is "src1". For regular receivers, this should always be equal to the topic name associated with the creation of the receiver object. However, with wildcard receivers, this indicates the actual source topic matched. The report also indicates the number of application messages successfully received, as well as the number of unrecoverable loss and unrecoverable burst loss events delivered to the application. Finally, the total number of message datagrams unrecoverably lost is shown. Note that this might not be equal to the number of application messages lost - if the publisher sends large messages that exceed the configured maximum datagram size, those large messages will be fragmented into multiple datagrams, each one of which is given a sequence number. The UM receiver has no way of knowing if a given set of umrecoverably lost datagrams represent multiple application messages or one large message.

Tmon: connection delete

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:10 2021 and 418792 usec

tmon\_conn=7

msg\_count=1

loss\_count=0

burst\_count=0

dgrams\_lost=0

This report indicates that the application closed connection 7. The message and loss counts should be identical to the EOS counters. (But it is possible for a connection to close without an EOS, so both reports contain the counts.)

Tmon: receiver delete

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:11 2021 and 494361 usec

tmon\_rcv=3

This report indicates that the application closed receiver 3.

Tmon: source delete

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:05 2021 and 492973 usec

tmon\_src=6

This report indicates that the application closed source 6.

Tmon: context delete

app\_id=tmon\_example\_1

ip=192.168.1.20

pid=66824

tmon\_ctx=1

time=Sun Jan 3 07:28:36 2021 and 689022 usec

This report indicates that the application closed context 1.

Context statistics received from tmon\_example\_1 at 192.168.1.20, process ID=15b61, object ID=7ff22483f000, context instance=a5b2a48ec7dbd9b4, domain ID=0, sent Sun Jan 3 15:36:57 2021

Topic resolution datagrams sent : 0

Topic resolution datagram bytes sent : 0

Topic resolution datagrams received : 0

Topic resolution datagram bytes received : 0

Topic resolution datagrams dropped (version) : 0

Topic resolution datagrams dropped (type) : 0

Topic resolution datagrams dropped (malformed) : 0

Topic resolution send failures : 0

Topics in source topic map : 0

Topics in receiver topic map : 0

Unresolved topics in receiver topic map : 0

Unknown LBT-RM datagrams received : 0

Unknown LBT-RU datagrams received : 0

Number of times message send blocked : 0

Number of times message send returned EWOULDBLOCK : 0

Number of times response send blocked : 0

Number of times response send returned EWOULDBLOCK : 0

Number of duplicate UIM messages dropped : 0

Number of UIM messages received without stream info: 0

These are the automatic transport monitoring statistics for the application's context. They are sent periodically by an internal timer.

Source statistics received from tmon\_example\_1 at 192.168.1.20, process ID=15b61, object ID=7ff22483f000, context instance=a5b2a48ec7dbd9b4, domain ID=0, sent Sun Jan 3 15:37:02 2021

Source: LBTRM:192.168.1.20:12090:db1fd69:239.101.3.11:12000

Transport: LBT-RM

LBT-RM datagrams sent : 0

LBT-RM datagram bytes sent : 0

LBT-RM datagrams in transmission window : 0

LBT-RM datagram bytes in transmission window : 0

LBT-RM NAK packets received : 0

LBT-RM NAKs received : 0

LBT-RM NAKs ignored : 0

LBT-RM NAKs shed : 0

LBT-RM NAKs ignored (retransmit delay) : 0

LBT-RM retransmission datagrams sent : 0

LBT-RM datagrams queued by rate control : 0

LBT-RM retransmission datagrams queued by rate control: 0

LBT-RM retransmission datagram bytes sent : 0

These are the automatic transport monitoring statistics for the applications sources. Note that this represents one transport session, which can have multiple topics mapped to it by the publisher. Also note that, unlike the TmonConnection creation report, the source string does not include a topic index in brackets. The data shown here is aggregated across all topics on the transport session.

Receiver statistics received from tmon\_example\_1 at 192.168.1.20, process ID=15b61, object ID=7ff22483f000, context instance=a5b2a48ec7dbd9b4, domain ID=0, sent Sun Jan 3 15:37:02 2021

Source: LBTRM:192.168.1.20:12090:db1fd69:239.101.3.11:12000

Transport: LBT-RM

LBT-RM datagrams received : 0

LBT-RM datagram bytes received : 0

LBT-RM NAK packets sent : 0

LBT-RM NAKs sent : 0

Lost LBT-RM datagrams detected : 0

NCFs received (ignored) : 0

NCFs received (shed) : 0

NCFs received (retransmit delay) : 0

NCFs received (unknown) : 0

Loss recovery minimum time : 18446744073709551615ms

Loss recovery mean time : 0ms

Loss recovery maximum time : 0ms

Minimum transmissions per individual NAK : 18446744073709551615

Mean transmissions per individual NAK : 0

Maximum transmissions per individual NAK : 0

Duplicate LBT-RM datagrams received : 0

LBT-RM datagrams unrecoverable (window advance) : 0

LBT-RM datagrams unrecoverable (NAK generation expiration): 0

LBT-RM LBM messages received : 0

LBT-RM LBM messages received with uninteresting topic : 0

LBT-RM LBM requests received : 0

LBT-RM datagrams dropped (size) : 0

LBT-RM datagrams dropped (type) : 0

LBT-RM datagrams dropped (version) : 0

LBT-RM datagrams dropped (hdr) : 0

LBT-RM datagrams dropped (other) : 0

LBT-RM datagrams received out of order : 0

These are the automatic transport monitoring statistics for the applications receivers. Note that this represents one transport session, which can have multiple topics mapped to it by the publisher. Also note that, unlike the TmonConnection creation report, the source string does not include a topic index in brackets. The data shown here is aggregated across all topics on the transport session.

Also note the very large numbers shown for "Loss recovery minimum time" and "Minimum transmissions per individual NAK". Those numbers are an unsigned representation of "-1", and mean that, as of that record, there was no NAK data on which to set min/max/avg statistics.

**Appendix 1 -**

**Integrating with an Already-Existing Source Notification Function**

Section 5.3, "TmonReceiver", is written assuming that your application does not currently use the UM Source Notification Function callbacks. If your application does make use of it, then you need to integrate Tmon connection management objects with your existing usage.

The primary purpose for the callbacks is to allow the application to create a per-source state structure or object when the delivery controller is created, and dispose of that state when the delivery controller is deleted. The configured creation method returns an object pointer to UM, and UM passes that same pointer to the UM receiver application callback via  
"msg.sourceClientObject()".

For Tmon to work correctly, the source notification callback must have access to the TmonReceiver object, so it should be passed as part of the application's client data object, set in the source notification function configuration.