WebSphere MQ Low Latency Messaging in Financial Exchanges

- Ultra high-speed and low latency messaging among critical components
- High availability for components found in a financial exchange
- Persistent messages, events with Coordination Manager

Roger Wong
Pablo B. Sigüenza González
Marco A Rojas
Chao Zhang

ibm.com/redbooks
Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (August 2011)

This edition applies to WebSphere MQ Low Latency Messaging v2.6.

This document created or updated on August 12, 2011.
# Contents

**Notices** ............................................................. vii  
**Trademarks** .......................................................... viii  

**Preface** .............................................................. ix  
The team who wrote this paper ........................................ x  
Now you can become a published author, too! .......................... xi  
Comments welcome ...................................................... xii  
Stay connected to IBM Redbooks ....................................... xii  

**Chapter 1. Introduction and concept.** ........................... 1  
1.1 Requisite knowledge ............................................... 2  
1.2 IBM WebSphere MQ Low Latency Messaging review ............. 2  
1.3 Reliable Multicast Messaging ..................................... 3  
1.3.1 Reliable Multicast Messaging API ............................ 4  
1.3.2 RMM support for TCP communication ......................... 5  
1.4 High Availability with Reliable and Consistent Messaging Streaming .................................................. 5  
1.4.1 Reliable and Consistent Message Streaming API .......... 6  
1.4.2 RCMS and message persistence ............................... 6  
1.4.3 RCMS use case .................................................... 7  

**Chapter 2. The demonstration environment.** .................... 9  
2.1 System setup ....................................................... 10  
2.1.1 Linux systems .................................................... 10  
2.1.2 WebSphere MQ Low Latency Messaging product .......... 21  
2.1.3 Uninstall .......................................................... 26  
2.2 Sample components ................................................ 27  
2.2.1 Order Submission Client .................................... 29  
2.2.2 Order Gateway .................................................. 30  
2.2.3 Matching Engine ................................................. 31  
2.2.4 Market Data Publisher ....................................... 32  
2.2.5 Packet Format Converter ................................... 33  
2.2.6 Message Store .................................................... 34  
2.2.7 Retransmission Service ...................................... 35  
2.2.8 Client Subscriber ................................................ 36  

**Chapter 3. Developing highly available matching engines with RCMS** ............................................................ 37  
3.1 A highly available matching engine ............................ 38  
3.1.1 Considerations .................................................... 38  
3.1.2 Overview of the sample application ......................... 39  
3.2 Setting up an RCMS tier ......................................... 42  
3.2.1 RCMS configuration file ..................................... 42  
3.2.2 LLM parameters for RCMS ................................. 43  
3.2.3 RCMS in C/C++ .................................................... 44  
3.3 Events handling .................................................... 46  
3.3.1 Change of member role callback ............................ 46  
3.3.2 State synchronization callbacks ............................. 47  
3.4 Message order ..................................................... 48  
3.4.1 Total order ....................................................... 49  
3.4.2 Modes of synchrony ............................................. 50
Related publications ................................................................. 99
IBM Redbooks publications .................................................. 99
Online resources ................................................................. 99
Help from IBM ................................................................. 100
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

<table>
<thead>
<tr>
<th>AIX®</th>
<th>Passport Advantage®</th>
<th>WebSphere®</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataPower®</td>
<td>Redbooks®</td>
<td>xSeries®</td>
</tr>
<tr>
<td>IBM®</td>
<td>Redpaper™</td>
<td></td>
</tr>
<tr>
<td>ILOG®</td>
<td>Redbooks (logo) (®)</td>
<td></td>
</tr>
</tbody>
</table>

The following terms are trademarks of other companies:

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.
Preface

This IBM® Redpaper™ publication demonstrates the application of IBM WebSphere® MQ Low Latency Messaging in financial exchanges and brokerages to facilitate ultra high-speed and low latency messaging among critical components such as order entry gateways and matching engines in a highly available fashion.

In this publication, we explain the environment used for our scenario and provide an explanation of the components developed and tested. We demonstrate high availability obtained with WebSphere MQ Low Latency Messaging utilized by some of the most critical components found in a financial exchange.

We illustrate how to extend our sample application to create a highly available matching engine component using the RCMS API. In addition, we discuss how the state of the application is transferred from the leading tier member when a tier member joins the tier. We demonstrate the RCMS features used to keep messages in the same order, so every tier member receives exactly the same input, which guarantees the same output from the tier. We also explain how to use real synchrony, virtual synchrony, and fast real synchrony, which is only available in LLM 2.6.

We cover the Coordination Manager Server, which is a component of WebSphere MQ Low Latency Messaging that provides a complementary message persistence solution to store messages and events into the data space. The Coordination Manager Server used in conjunction with the client interface libraries (LLMI) provide the ability to store, retrieve, and replay messages from persistent storage.

This publication was designed for an audience of solution architects and chief technical officers (CTOs) in the Financial Sector marketplace, application developers starting out with WebSphere MQ Low Latency Messaging, and the wider messaging community (WMQ and so forth).
This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Roger Wong is a Senior Solutions Architect for WebSphere Front Office Financial Service Solution and MQ Low Latency Messaging. He has over 15 years of systems software design and development experience. Roger worked as Systems Architect in a major brokerage firm prior to joining IBM in 2007. His areas of expertise include market data and trading systems, and relational database designs. He holds a B.S. degree in Computer Science from Georgia Institute of Technology.

Pablo B. Sigüenza González is a Senior Architect who has over 10 years of experience in JEE technologies. He joined InterWare de Mexico in 2007 and he is in charge of the Training and Research Office which is mainly focused on research on new tools such as MQ Low Latency Messaging. His areas of expertise include several WebSphere products as IBM DataPower®, Message Broker, Process Server, IBM ILOG® JRules, MQ LLM, and Lombardi, as well other JEE technologies. Pablo has the following IBM certifications: IBM Certified Associate BPM Analyst - BPM Blueprint; WebSphere Lombardi Edition V7.1, IBM Certified Application Developer - WebSphere ILOG JRules V7.x, IBM Certified Solution Implementer - WebSphere DataPower SOA Appliances Firmware V3.7.3, IBM Certified Solution Developer - WebSphere Message Broker V6.1, IBM Certified Application Developer - WebSphere ILOG JRules 6.x, IBM Certified System Administrator - WebSphere Application Server, Network Deployment V6.1, and IBM Certified System Administrator - WebSphere MQ V6.0.
Marco A Rojas is a Senior System Administrator and Senior Software Developer for Mexican Stock Exchange. He has an extensive background in software development, support, and distributing system technologies with over 8 years of experience. His skills encompass the following areas: Enterprise Application Server (webLogic, jbossas, WebSphere), Enterprise Web Server (sjsws, httpd), Enterprise Messaging (sonicmq, jboss messaging, activemq), LDAP (sjsdsee, openldap), and Enterprise Software and Web Development (J2EE). Marco is RHCT/RHCSA certified.

Chao Zhang is a Staff Software Engineer who is working on the IBM MQ Low Latency Messaging L3 support team. He received his Masters degree in Computer Science from the Electronics and Science University in Xi’an, China. He has over 7 years of experience in distributing system development. He worked as a senior software engineer in the grid computing industry prior to joining IBM. His areas of expertise include telecommunications and the cloud computing industry.

Thanks to the following people for their contributions to this project:

Tamikia Barrow, Shari Deiana
International Technical Support Organization, Raleigh Center

Arnaud Mathieu, IBM Software Group, Application and Integration Middleware Software, Program Director, Financial Services Solution Development
IBM Austin

Chris Kime, IBM Software Group, Application and Integration Middleware Software, Application Development
IBM Austin

Rufus P. Credle Jr., ITSO GCS Project Leader, Sales and Distribution
IBM Raleigh

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at: ibm.com/redbooks/residencies.html
Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- Use the online Contact us review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:
  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:
  http://twitter.com/ibmredbooks
- Look for us on LinkedIn:
  http://www.linkedin.com/groups?home=&gid=2130806
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
- Stay current on recent Redbooks publications with RSS Feeds:
  http://www.redbooks.ibm.com/rss.html
Introduction and concept

In this chapter, we demonstrate the application of WebSphere MQ Low Latency Messaging in financial exchanges and brokerages to facilitate ultra high-speed and low latency messaging among critical components, such as order entry gateways and matching engines, in a highly available fashion.

This paper further extends the use of WebSphere MQ Low Latency Messaging in an information distribution system where matching engine outputs are published and persisted using WebSphere MQ Low Latency Messaging.
1.1 Requisite knowledge

To take advantage of the contents of this paper, you need to meet the following requirements:

- Programming experience:
  The examples included in this document are written in C/C++ or Java. You need to be comfortable with those languages.

- WebSphere MQ Low Latency Messaging (WMQ-LLM) experience:
  In preparation, first read the *WebSphere MQ Low Latency Messaging Development Guide*, ZG24-1775-00 in order to obtain more detailed information about the concepts, features, and techniques of WMQ-LLM.

- Networking experience:
  You need to be familiar with the following terms:
  - Networking components:
    - Local Area Network (LAN)
    - Wide Area Network (WAN)
    - Switch
    - Router
    - Ethernet
    - Network Interface Card (NIC)
  - Networking protocols:
    - Internet Protocol (IP)
    - Transmission Control Protocol (TCP)
    - User Datagram Protocol (UDP)
    - IP multicast

1.2 IBM WebSphere MQ Low Latency Messaging review

IBM WebSphere MQ Low Latency Messaging is a high-throughput, low-latency transport fabric designed for one-to-one, one-to-many, or many-to-many data exchange in a message-oriented middleware publish and subscribe fashion.

Applications include the high-speed delivery of market data, transactional data, reference data, and event data in or between front-, middle-, and back-office.

Although initially designed to meet the high-speed and throughput requirements of financial services firms, WebSphere MQ Low Latency Messaging is suitable for use by other industries with similar requirements. Characteristic applications require extremely low latency and high message volumes (ranging from many thousands to millions of messages per second), with positive or negative acknowledgement reliability.

WebSphere MQ Low Latency Messaging exploits the Internet Protocol (IP) multicast infrastructure to ensure scalable resource conservation and timely information distribution. Both Ethernet and InfiniBand networks are supported.

Reliability and traffic control are added on top of high performance User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) networking implementations. WebSphere MQ Low Latency Messaging takes this one step further and enables the support of highly available unicast or multicast data distribution, by implementing a number of stream failover policies that allow seamless migration of transmission from failed to backup processes.
In summary, WebSphere MQ Low Latency Messaging offers the following advantages:

- Multiple and flexible message transports
- Reliability
- High availability and reliability in a tiered, replicated environment using WebSphere MQ Low Latency Messaging
- Reliable and Consistent Message Streaming (RCMS) functionality:
  - Component replication, automatic failure detection, and failover
  - Total order sequencing
  - Automatic state synchronization for joining applications
  - Single acknowledgment multicast reliability
  - Support for multiple levels of redundancy, with inter-tier communications
  - Support for both duplex and replay modes
- Message filtering
- Monitoring
- Congestion management
- High performance
- Support for multiple platforms and programming languages


1.3 Reliable Multicast Messaging

Reliable Multicast Messaging (RMM) is one of the two protocols supported by WebSphere MQ Low Latency Messaging. It implements different levels of reliability on top of a standard unreliable networking infrastructure. This includes the UDP and IP multicasting protocols. RMM will also work over InfiniBand and shared memory.

RMM can operate in either multicast (one-to-many) or unicast (one-to-one) mode.

RMM achieves higher levels of reliability (over standard multicast) by buffering outgoing and incoming data. The total memory footprint of RMM and individual buffer sizes are configurable:

- Buffering incoming data tolerates lost messages or spikes in message traffic that can overflow system buffers when an application is unable to process messages at the incoming rate.
- Buffering outgoing data in the history buffer tolerates traffic spikes that can overwhelm system buffers and facilitates the retransmission of data missed by receiver applications.

RMM supports both negative acknowledgement (NAK) and positive acknowledgement (ACK) modes of receiver feedback:

- NAK mode is supported on both multicast and unicast topics. It allows receivers to request a retransmission of missed data from the transmitter's history buffer. Thus, the transport's reliability is dependent on the availability of the required messages in the transmitter's history buffer.
- ACK mode requires a positive acknowledgement of each packet the receiver has received, prior to the packet being discarded from the transmitter's history buffer.
Variations of ACK mode can also be used. These are referred to as WAIT1-ACK and WAITN-ACK mode.

**Reference:** See Chapter 3 in the *WebSphere MQ Low Latency Messaging Development Guide*, ZG24-1775-00.

### 1.3.1 Reliable Multicast Messaging API

In this paper, we provide sample code with our examples. The RMM APIs that are covered in the sample code are listed in Table 1-1, Table 1-2, and Table 1-3.

<table>
<thead>
<tr>
<th>Table 1-1</th>
<th>RMM Callbacks methods used in samples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Callback methods</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>on_message</td>
<td>Every time a message is received on the topic for which this callback was registered, <em>on_message</em> is invoked.</td>
</tr>
<tr>
<td>on_event</td>
<td>This method is called whenever a stream event occurs on the topic. This callback is used for both forwarder and received topics, and all senders topics.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1-2</th>
<th>RMM Receiver APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Receiver methods</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>rmmRxCreateTopic</td>
<td>Create a receiver regular topic</td>
</tr>
<tr>
<td>rmmRxInit</td>
<td>Initialize the receiver instance</td>
</tr>
<tr>
<td>rmmRxJoinMulticastGroup</td>
<td>Join the multicast group first then subscribe to the topics for this multicast group. Prerequisite to creating the receive topic and start receiving messages.</td>
</tr>
<tr>
<td>rmmRxStop</td>
<td>Stop the receiver instance. Some delay is expected for thread termination and cleanup.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1-3</th>
<th>RMM Transmitter APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transmitter methods</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>rmmTxCreateTopic</td>
<td>Create a transmitter regular topic.</td>
</tr>
<tr>
<td>rmmTxInit</td>
<td>Initializes the transmitter instance.</td>
</tr>
<tr>
<td>rmmTxStop</td>
<td>Stop the sender instance. Some delay is expected for thread termination and cleanup.</td>
</tr>
<tr>
<td>rmmTxSubmitMsg</td>
<td>The message send API used to send a message or forward a received message.</td>
</tr>
</tbody>
</table>
1.3.2 RMM support for TCP communication

As mentioned before, RMM was originally developed to provide reliable delivery of multicast data using unreliable UDP protocol.

This is usually fine when all the components are over a LAN, but if you need to connect to a remote component over a WAN, that is a completely different story. The main issue is that most of the time, multicast traffic is not allowed over the WAN because special routers and network configuration setup are required. Network administrators generally disallow this kind of traffic.

The solution to the foregoing scenario is to use RMM over TCP instead of UDP protocol.

RMM support for TCP communication allows you to use the following modes:

- Unicast over TCP/IP
- Unicast over native InfiniBand

The main advantage of supporting TCP with RMM is that all functions available in RMM API are supported over TCP. That means you can reuse most of your existing RMM code to support all protocols and combine them within a single RMM instance or a Multicast by Unicast (MBU) topic.

RMM-TCP is intended to provide TCP-like support to an existing RMM application. Minimum code change is needed to add RMM-TCP support to an application developed with RMM API. WebSphere MQ Low Latency Messaging provides true TCP-like bidirectional messaging and flow control through Reliable Unicast Messaging (RUM) API. RUM is more efficient than traditional TCP, therefore, well suited as a TCP replacement for the connection over a WAN.

1.4 High Availability with Reliable and Consistent Messaging Streaming

Reliable and Consistent Message Streaming (RCMS) is a part of Low Latency Messaging (LLM) that provides high availability and consistent, ordered delivery. RCMS enhances the high availability capabilities of RMM. RCMS uses component replication for high availability. This means that multiple instances of the application run simultaneously. RCMS ensures that all instances of the application get exactly the same input, and can therefore maintain identical state. If one of the instances of the application fails, RCMS detects the failure and automatically takes corrective actions. RCMS also provides the service of synchronizing a new instance of the application (for example, to replace an instance that failed).

RCMS supports the following features:

- Total ordering
- Real-synchrony and virtual-synchrony tier
- Fast failure detection and failover
- New member synchronization
- Management of failover streams
- Intra-tier communication
- Support of data-source restart
- Support for multi-site failover
- Full access to all RMM APIs monitoring
1.4.1 Reliable and Consistent Message Streaming API

Reliable and Consistent Message Streaming (RCMS) is built as an extension to RMM so that most the RMM APIs (monitoring, congestion management, and so on) can be used while working with RCMS. The application can use a combination of RCMS Tier topics and regular RMM topics that are created on the same RCMS instance.

The RCMS API that is covered in the sample code is shown in Table 1-4.

Table 1-4   RCMS APIs

<table>
<thead>
<tr>
<th>RCMS methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmmTierCreateRxTopic</td>
<td>Create a receiver tier topic</td>
</tr>
<tr>
<td>rmmTierCreateTxTopic</td>
<td>Create a transmitter tier topic.</td>
</tr>
<tr>
<td>rmmTierInit</td>
<td>Initialize the tier instance</td>
</tr>
<tr>
<td>rmmTierStop</td>
<td>Stop the tier instance. Some delay is expected for thread termination and cleanup.</td>
</tr>
<tr>
<td>rmmTierSubmitMessage</td>
<td>The message send API used to send a message or forward a received message using RCMS.</td>
</tr>
</tbody>
</table>

1.4.2 RCMS and message persistence

Many highly available systems use message persistence where messages are persisted to some form of reliable storage. In such systems, part of the failure recovery mechanism is to retrieve the messages from storage.

The important thing to note is that RCMS does not use any form of message persistence to provide HA services. Instead, RCMS uses component replication where each instance of the application is replicated multiple times, which makes the application resilient to failures. The main advantage of the approach taken by RCMS is in the low latency and high throughput it facilitates; other advantages include faster recovery time and greatly simplifying the involvement of the application in HA aspects.

Although RCMS does not use message persistence for HA, an application might want to persist messages to storage for other reasons (for example, to persist data for legal purposes). An application that wants to persist the messages it processes can easily do this with the WebSphere MQ Low Latency Messaging Message Store feature.

For more information regarding LLM, visit the Information Center at the following Web page: http://publib.boulder.ibm.com/infocenter/wllm/v2r5pwd/topic/com.ibm.wllm.doc/rcms_ug.html

Access key: If you have purchased WebSphere MQ Low Latency Messaging Version 2.5 or 2.6, you can request an access key from the following link to enable you to view the aforementioned page within the Information Center.
1.4.3 RCMS use case

Figure 1-1 describes a typical implementation of a highly available IT infrastructure using RCMS in the financial market industry.

The scenario depicted in this paper includes three major components:

- Matching engine (ME)
- Gateway (GW)
- An order entry application

Figure 1-1 provides a description of the operations performed by these components.

- Order entry application: A proxy application (Brokers/Clients) sends orders to the gateway through LLM RMM-TCP API.
- Order Gateway: Receives orders through LLM RMM API and routes them to the appropriate Matching Engine (ME) through RCMS. On the return path, the gateway (GW) receives Responses (ACK/Trade) from the ME and sends them to the Clients and Brokers.
- Matching Engine: Receives orders through RCMS from multiple GWs and performs order matching. Sends processed orders and trades to the appropriate GWs and logs them into storage using WMQ-LLM message store. Data/messages can be retrieved using WMQ-LLM the message store.

Figure 1-1  RCMS scenario
The demonstration environment

In this chapter, we explain the environment that we used and documented in this paper, as well as providing a brief explanation of the components developed and tested.

Our main objective is to demonstrate the high availability obtained with WebSphere MQ Low Latency Messaging (WMQ-LLM) as it was working with some of the most critical components found in a financial exchange.
2.1 System setup

WebSphere MQ Low Latency Messaging is a scalable product. Performance depends on the complexity of the specific environment, volume of data traffic, and the data object size. The actual requirements for your system might be different, depending on the complexity of your specific environment, latency, throughput, and data requirements.

The following additional hardware might be required based on capacity requirements:

- CPUs (processors or multicore processors)
- Memory
- Servers (for added capacity or redundancy)
- Disks

Gigabit network adapters and routers are required for high-speed transport. Native InfiniBand (on Linux only) and IP over InfiniBand are also supported.

The actual throughput that any application will experience can vary, depending upon considerations such as message size, transmission rate, hardware platform, and network configuration. Therefore, no assurance can be given that an individual application will achieve the throughput or latency stated here. Customers must conduct their own testing. For more detailed performance information, consult your IBM representative.

In the following sections, we describe the environment built in our lab to produce the content represented in this paper.

Reference: For details about supported hardware and software, see the IBM US Announcement Letter 211-074 at the following website:

2.1.1 Linux systems

In order to have a homogeneous environment, all the machines were running Red Hat Enterprise Linux AS 5 update 4 32 bit (Table 2-1).

Table 2-1   Operating system used

| Platform | RHEL 5.4 (Kernel 2.6.18-164.el5PAE) i386 |

Reference: WMQ-LLM 2.6 is supported over different operating system platforms. See the ENUS211-074 for detailed information regarding this point.

Compiler

We used the GNU gcc version 4.1.2 (20080704), all the components we developed for this paper were done by using C/C++

Reference: For a full reference about “Runtime requirements” and “Development system software requirements,” see the ENUS211-074.
Kernel parameters

The Linux kernel is flexible, and you can even modify the way it works on the fly by dynamically changing some of its parameters. Although this flexibility is very useful when you are doing tests, after reboot you will lose all the parameters made at runtime, therefore you need to consider persisting your changes across reboots.

Linux does a good job of auto-tuning the TCP buffers, but the default maximum Linux TCP buffer sizes are too small, so we used the following configuration shown in Example 2-1.

Example 2-1  Kernel settings for network

```
$ cat /etc/sysctl.conf
...
# redp4752
net.core.rmem_max = 4194304
net.core.wmem_max = 4194304
net.core.rmem_default = 4194304
net.core.wmem_default = 4194304
net.ipv4.tcp_rmem = 8192 4194304 8388608
net.ipv4.tcp_wmem = 8192 4194304 8388608
net.ipv4.tcp_mem = 4194304 4194304 8388608
net.ipv4.igmp_max_memberships = 30
net.core.netdev_max_backlog=3000
```

On InfiniBand networks, we have the following configuration shown in Example 2-2.

Example 2-2  Kernel settings for an InfiniBand network

```
$ cat /etc/sysctl.conf
...
net.core.rmem_default = 65536
net.core.wmem_default = 65536
net.core.rmem_max = 8388608
net.core.wmem_max = 8388608
# swg21450934
net.ipv4.conf.all.arp_ignore=1
net.ipv4.conf.ibm.arp_ignore=1
net.ipv4.conf.ibi.arp_ignore=1
```

Runlevel

Normally, there is no need for a GUI on a Linux server (usually enabled in runlevel 5). All administration tasks can be achieved by the command line, redirecting the X display or through a Web browser interface, then you can manually disable those services or you can modify the `inittab` file to set boot level as 3. See Example 2-3.

Example 2-3  Server will boot at runlevel 3

```
$ cat /etc/inittab
...
# The default runlevel is defined here
```
id:3:initdefault:

...

$ runlevel # shows the last and current runlevel

Services
Every non-essential service that is running causes scheduling delays and takes away CPU
time from the high speed messaging application. Disable unnecessary services, processes,
and threads in the system, including the irqbalance services. It will reduce the possibility that
other services, processes, or threads will replace the LLM related ones and negatively affect
the messaging performance.

The following list is a small subset of the common Linux services that need to be stopped if
they are not critical to the operational environment:

- sendmail
- nfslock
- netfs
- isdn
- bluetooth
- cups
- autofs
- xfs

Disabling services: To check and disable the services running on a specific runlevel, you
can use the following commands; we assume a system run level 3:

```
# /sbin/chkconfig --list | grep 3:on # check all services running
# /sbin/chkconfig --level 3 sendmail off # turn off service
```

User limits
Linux provides control over the resources available to the user shell and to processes started
by it. Sometimes these limits can prevent optimum performance for user processes after a
maximum limit value is reached. In this case, you are able to modify those user limits to
achieve a better performance depending of the process behavior.

Based on our experience, we have tuned the file descriptors for these tests. See
Example 2-4).

```
Example 2-4   Limits for user shell

$ id
uid=502(redpaper) gid=502(redpaper) groups=502(redpaper) ...
$ cat /etc/security/limits.d/redp4752.conf
...
# REDP4752 - allow the use of 65535 open files
redpaper    soft nofile  65535
redpaper    hard nofile  65535

$ ulimit -a # show user limits
```

Checking files: At runtime you can check the open files with the following command:

```
# cat /proc/sys/fs/file-nr
```
When using the native InfiniBand implementation, it might be necessary to increase the memlock limit. This is required to allow the OFED implementation to be able to pin a large amount of memory. See Example 2-5.

Example 2-5  Set memlock for InfiniBand

$ cat /etc/security/limits.d/redp4752.conf
...
# REDP4752 - set memlock to 256MB
redpaper    soft    memlock 262144
redpaper    hard    memlock 262144

$ ulimit -a # show user limits

Environment variables
At the process initialization, it is highly desirable to have good control over the environment variables for the process. You can decide whether it is better to have control for those variables on every script or in one single profile.

For our test, we chose to use a profile, because all the components use the same environment variables. See Example 2-6.

Example 2-6  Profile for user session

# /etc/profile.d/redp4752.sh - set REDP4752 environment stuff

HISTCONTROL=ignoreboth
HISTSIZE=5000
HISTFILESIZE=5000
HISTTIMEFORMAT='%F %T '

MAINLLMPROCESS=CLIENT # main name component used on server here!
[ $(/usr/bin/id -u) != 0 ] && PS1='[$u@h:$MAINLLMPROCESS \W]\$ '

if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi

LLMHOME="/opt/ibm/llm"

OS_ST=32
[ "$(uname -m)" = "x86_64" ] && OS_ST="64"

LLMLIB="$LLMHOME/lib${OS_ST}"

if [ -n "${LD_LIBRARY_PATH}" ] ; then
    LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${LLMLIB}"
else
    LD_LIBRARY_PATH="${LLMLIB}"
fi
export LLMHOME LD_LIBRARY_PATH

EDITOR=vim
export EDITOR

ulimit -c unlimited
set -o vi
Clock synchronization

In order to maintain reliability for all components and latency metrics, it is required to have a good clock synchronization on the systems where LLM components are running.

An important note here is that depending on the accuracy in the clock time, you can obtain different latency metrics on messages crossing different servers. Because most common clock synchronization methods give accuracy in milliseconds, you can have unreliable differences on latency metrics with LLM components as they work with messages at microsecond or nanosecond rates, especially when working with 10G or InfiniBand networks.

A reliable round trip latency measurement strategy is to have a message reply so that start and end time can be taken on the same machine and therefore using the same clock. It is necessary to have a better clock synchronization at microsecond or nanosecond level if you want to get reliable latency metrics with distributed components.

For purposes of the lab, we used an NTP server to synchronize servers clocks. We worked with round trip latency and all phases of latency metrics. Those metrics with sample components are covered in Chapter 5, “Test scenarios” on page 73.

Example 2-7 shows the `ntpstat` command output.

```
Example 2-7   Command to show clock synchronization on machine

$ ntpstat
synchronised to NTP server (66.187.233.4) at stratum 2
time correct to within 31 ms
polling server every 1024 s
```

Times: Usually larger round-trip times (RTTs) are correlated to higher delay variability, but this needs to be weighed against the accuracy of nearby clocks versus remote clocks (if nearby are stratum 4, and remote are stratum 1).

Network adapters

Gigabit network adapters and switches are required for high-speed transport. A good network adapter (NIC, HCA) with a recent device driver chosen by the OS provider is the base for good performance.

Ensure that you have the latest firmware version for your network switches and Network Interface Cards (NICs). This applies also to the InfiniBand switches and host channel adapter (HCA) cards. Often there are key bug fixes especially in the area of IP multicast that might be absent in older firmware versions of network gear.

You can check the firmware versions of your network adapters with the following commands. See Example 2-8.

```
Example 2-8   Check firmware versions of adapters

# ethtool -i eth0 # NIC
driver: bnx2
version: 1.6.7c
firmware-version: 4.0.3
```
GbE adapters

Gigabit-based network interfaces have many performance-related parameters inside of their device driver such as CPU affinity. Also, the TCP protocol can be tuned to increase network throughput, some other concepts such as interrupt rate, buffering, and enabling TCP offloading features at NICs can help you to achieve a better performance.

The kernel module used in this paper for standalone GbE adapters was tigon3, for bonded GbE adapters was e1000. You can check the kernel module used for a specific NIC as shown in Example 2-9.

Example 2-9   Kernel module used for GbE adapter

```bash
$ dmesg | grep eth0
eth0: Tigon3 [partno(BCM95721) rev 4101 PHY(5750)] (PCI Express) 10/100/1000Base-T Ethernet 00:11:25:3f:18:7a
tg3: eth0: Link is up at 1000 Mbps, full duplex.
tg3: eth0: Flow control is on for TX and off for RX.

$ dmesg | grep eth2
e1000: eth2: e1000_probe: Intel(R) PRO/1000 Network Connection ADDRCONF(NETDEV_UP): eth2: link is not ready
```
InfiniBand adapters

The InfiniBand Architecture (IBA) is an industry standard that defines a new high-speed switched fabric subsystem designed to connect processor nodes and I/O nodes to form a system network.

This interconnect method moves away from the local transaction-based I/O model across buses to a remote message-passing model across channels. The architecture is independent of the host operating system and the processor platform.

IBA provides both reliable and unreliable transport mechanisms in which messages are queued for delivery between end systems. Hardware transport protocols are defined that support reliable and unreliable messaging (send/receive), and memory manipulation semantics (for example, RDMA read/write) without software intervention in the data transfer path.

The LLM native InfiniBand implementation supports the hardware and software supported by OpenFabrics Enterprise Distribution (OFED). LLM uses the InfiniBand verbs interface from OFED for transmitting and receiving messages. RMM uses unreliable datagram queue pairs (UD QPs) and RUM uses reliable connected queue pairs (RC QPs) for transmitting and receiving messages.

IPoIB comes inside an OFED software stack, which is a network driver implementation that enables transmitting IP and ARP protocol packets over an InfiniBand UD channel.

The InfiniBand rdmacm interface from OFED is also required for connection management and multicast group join and leave operations.

OFED: The following website provides instructions for downloading OFED:
http://www.openfabrics.org/downloads/OFED/

InfiniBand Drivers need to start at boot time. The OFED 1.5.2 comes with a service called openibd that manages activation and deactivation of kernel modules (Example 2-10).

Example 2-10  Kernel module used for InfiniBand adapter

```bash
$ /sbin/chkconfig --list | grep ib
openibd   0:off   1:off   2:on    3:on    4:on    5:on    6:off

$ /sbin/lsmod | grep ib
ib_ucm         50312  0
ib_sdp         193116  0
rdma_cm        77204  2 rdma_ucm,ib_sdp
ib_addr         43016  1 rdma_cm
ib_ipoib       114528  0
ipoib_helper    35728  2 ib_ipoib
ib_cm          71592  3 ib_ucm,rdma_cm,ib_ipoib
ib_sa          76424  4 rdma_ucm,rdma_cm,ib_ipoib,ib_cm
ib_uverbs      75824  2 rdma_ucm,ib_ucm
```
Multicast

Multicast IP is a protocol that allows transmission of packets to a defined group of recipients. This has the advantage of network level transmission of a single message from the sender to multiple receivers, which is much more efficient than handling multiple messages in code.

You need to make sure the multicast is available for your network segment. At the NIC, you can check this using the `ifconfig` command. Example 2-11.

```
Example 2-11   Multicast label enabled
# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:11:25:3F:18:7B
inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
inet6 addr: fe80::211:25ff:fe3f:187b/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500  Metric:1
RX packets:329420 errors:0 dropped:0 overruns:0 frame:0
TX packets:321027 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:40952432 (39.0 MiB)  TX bytes:39952757 (38.1 MiB)
Interrupt:169 Memory:c9ff0000-ca000000
```

Tip: If not available, use the following command to enable multicast:
```
# ifconfig <interface> multicast
```

Example 2-12   Multicast throughput test
```
$ iperf -s -B 239.255.1.3 -u -f m -i 5 # sender
$ iperf -c 239.255.1.3 -u -b 990m -f m -i 5 -t 30 # receiver
```

Tip: If you want to limit the multicast traffic, you have to add a dedicated route on the network device you have defined:
```
# route add -host 239.255.1.3 ethX
```
On Infiniband networks, multicast support is fabricated inside the switches using forwarding tables, which are set up by the Subnet manager. The nodes make explicit requests to create or join multicast groups; just 0xC000 to 0xFFF are valid multicast LIDs.

**Failover support for Ethernet**

Low Latency Messaging allows an RMM transmitter instance to use more than one Network Interface. The application can use a single RMM transmitter instance where different topics send data on different Network Interface Cards (NICs). This feature allows an application to easily send data over several networks.

The application can specify a list of supplemental interfaces in the transmitter's configuration and then choose one of these interfaces to be used by a Topic transmitter. By default, the list of supplemental interfaces is empty and only a single interface (defined in the MulticastInterface field of rmmTconfig) is used.

**IP bonding**

The Linux bonding driver provides method for aggregating multiple network interfaces into a single logical bonded interface for backup or round robin solution. The bonding needs to work with a minimum of two NICs.

Before attempting to bond two NICs, you need to verify the integrity and functionality of each NIC. You can check your interface status with `ethtool` (Example 2-13).

*Example 2-13  The ethtool output*

```bash
# ethtool eth2
Settings for eth1:
   Supported ports: [ TP ]
   Supported link modes:   10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
                           1000baseT/Half 1000baseT/Full
   Supports auto-negotiation: Yes
   Advertised link modes:  10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
                           1000baseT/Half 1000baseT/Full
   Advertised auto-negotiation: Yes
   Speed: 1000Mb/s
   Duplex: Full
   Port: Twisted Pair
   PHYAD: 1
   Transceiver: internal
   Auto-negotiation: on
   Supports Wake-on: g
   Wake-on: g
   Current message level: 0x000000ff (255)
   Link detected: yes
```
For the purposes of this paper, we have used the active-backup mode with a link check interval of 100 milliseconds. After we have bonded two NICs, we expect to see the following results (Example 2-14).

**Example 2-14  Bonding configuration, 2 NIC used**

```
# cat /proc/net/bonding/bond0

Ethernet Channel Bonding Driver: v3.1.2 (January 20, 2007)

Bonding Mode: fault-tolerance (active-backup)
Primary Slave: None
Currently Active Slave: eth2
MII Status: up
MII Polling Interval (ms): 100
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth2
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:XX:XX:XX:XX:c2

Slave Interface: eth3
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:XX:XX:XX:XX:c4
```

**Tip:** You can use this command line to get more information about the bonding module:

```
# modinfo bonding
```

**Failover support for Infiniband**

Multiple NIC support can be used to provide high availability when using native InfiniBand as the transport mechanism. The multiple NIC support allows transparent failover to a separate physical InfiniBand HCA configured as a backup network interface.

There are three basic types of failures in an InfiniBand fabric: link failure, subnet manager failure, and switch failure:

- **Link failure**
  A link failure occurs when a specific HCA port fails but the rest of the fabric is unaffected.

- **Subnet manager failure**
  An InfiniBand subnet manager is responsible for establishing routes between the InfiniBand nodes in a fabric. If a subnet manager goes down and a standby subnet manager takes over, all the InfiniBand nodes in the fabric have to reregister with the new subnet manager.

- **Switch failure**
  A switch failure occurs when the entire InfiniBand switch fails affecting all the InfiniBand nodes connected to the switch. InfiniBand fabrics can be either connected, all InfiniBand nodes are logically connected on the same switch, or disconnected, no physical connection between InfiniBand subnets. Low Latency Messaging will take different recovery actions based on the type of failure.
Upon a failure of a physical link or InfiniBand network component, Low Latency Messaging will get immediate notification from the HCA by way of a local link-down event. Low Latency Messaging will be responsible for interpreting the event and re-establishing the affected queue pair (if necessary by switching traffic to a different local active port).

To support network failover in Native InfiniBand, the application has to provide each RMM transmitter topic or Multicast by Unicast (MBU) destination two interfaces to work with.

There are two modes in which the receiver can work to support redundant networks. The first is to duplicate the original set of rmmPorts on to an additional interface. This mode works well if each interface is connected to a different InfiniBand network and the networks are not connected. If the networks connected to the two interfaces are connected, you might want to consider working in the second mode where only one set of rmmPorts is used, but for each port, two interfaces are defined. By working in this mode, you can avoid getting duplicate data on two interfaces that are connected.

**IB bonding**

IB-bonding is a High Availability solution for IPoIB interfaces. It is based on the Linux Ethernet Bonding Driver and was adopted to work with IPoIB.

However, the support for IPoIB interfaces is only for the active-backup mode, other modes must not be used. To create an bonded interface configuration, you must use the standard syntax.

**Disk**

Most disk configurations are based on capacity requirements, not performance. This is a key factor when working with Low Latency Message Store because the speed of the disk impacts directly on its performance. It is highly desirable to make a sizing of storage and I/O performance at system design time.

There are two measures of disk speed commonly used: raw byte-transfer rate, and disk sync rate, the measure of speed is really how often the IO subsystem can sync data to disk (as opposed to the byte transfer rate).

Make sure that there is enough disk space to hold the messages that are to be stored. It is also desirable to increase storage speed and reliability through redundancy on your SAN volume or local disks.

In the test environment, we used 74G of internal SCSI storage on each server. You can perform timings of device reads for benchmark and comparison purposes with the `hdparm` command as shown in Example 2-15.

**Example 2-15  Perform timings of device reads**

```bash
# hdparm -Tt /dev/sda
/dev/sda:
Timing cached reads:  14386 MB in  1.99 seconds = 7212.38 MB/sec
Timing buffered disk reads: 312 MB in  3.01 seconds = 103.50 MB/sec
```

**Tip:** You can obtain more information of your storage device with the following command:

```bash
# hdparm -I /dev/sda
```
2.1.2 WebSphere MQ Low Latency Messaging product

The version of the WebSphere MQ Low Latency Messaging (WMQ-LLM) product that we used for this paper was 2.6.0.0. This version is available after April 6, 2011.

The product can be obtained using normal IBM software channels (IBM sales contact, or using IBM Passport Advantage®). The installation image for the product must be a physical CD or an ISO image of the CD.

**Tip:** Using Passport Advantage, the LLMv2 product can be downloaded as an ISO CD image with the product part number. The product part number can be found in the LLM Information Center:


The installation files for all supported platforms are contained on the same image.

**Tip:** The LLM product uses InstallAnywhere for the product installation. This presents an identical installation interface on any of the supported platforms, so the instructions are essentially identical for installing the product on all platforms.

Using the installation image acquired for the WMQ-LLM v2.6 product, perform the following steps to install the product on a Linux system.

**Tip:** If you are working with an ISO image on a Linux machine, you can mount the image locally using the `mount` command. For example, if you have copied the ISO image as the file `llm_disk_v26.iso` to the directory `/tmp`, you can mount the ISO image to `/mnt/cdrom` using the command:

```
# mount -o loop -t iso9660 llm_disk_v26.iso /mnt/cdrom
```

You can then access the files of the ISO image under the `/mnt/cdrom` path:

```
# ls /mnt/cdrom
```
Proceed as follows to perform the installation:

1. Execute the `install-Linux.sh` command (Figure 2-1) on the installation image; we will use English locale.

2. Read the introduction and click **Next** (Figure 2-2).
3. Accept the license agreement and click **Next** (Figure 2-3).

![Figure 2-3 Agreements for use LLM]

International Program License Agreement

**Part 1 - General Terms**

BY DOWNLOADING, INSTALLING, COPYING, ACCESSING, CLICKING ON AN "ACCEPT" BUTTON, OR OTHERWISE USING THE PROGRAM, LICENSEE AGREES TO THE TERMS OF THIS AGREEMENT. IF YOU ARE ACCEPTING THESE TERMS ON BEHALF OF LICENSEE, YOU REPRESENT AND WARRANT THAT YOU HAVE FULL AUTHORITY TO BIND LICENSEE TO THESE TERMS. IF YOU DO NOT AGREE TO THESE TERMS,

- DO NOT DOWNLOAD, INSTALL, COPY, ACCESS, CLICK ON AN "ACCEPT" BUTTON, OR USE THE PROGRAM; AND

- PROMPTLY RETURN THE UNUSED MEDIA, DOCUMENTATION, AND PROOF OF ENTITLEMENT TO THE PARTY FROM WHOM IT WAS OBTAINED FOR A REFUND OF THE AMOUNT PAID, IF THE PROGRAM WAS DOWNLOADED, DESTROY ALL COPIES OF THE PROGRAM.

Press Enter to continue viewing the license agreement, or enter "1" to accept the agreement, "2" to decline it, "3" to print it, or "99" to go back to the previous screen.

---

4. Specify the install location (Figure 2-4). We will use the default directory /opt/ibm/llm. Click **Next**.

![Figure 2-4 Folder destination]

Choose Install Folder

Choose a destination folder for this installation

Where would you like to install?

Default Install Folder: /opt/ibm/llm

ENTER AN ABSOLUTE PATH, OR PRESS <ENTER> TO ACCEPT THE DEFAULT : 
5. Accept the default (Figure 2-5) not to create links and click **Next**.

![Figure 2-5 Do not use links](image1)

6. Perform a customer installation, option 3. The default option does not have API for message store (Figure 2-6). Click **Next**.

![Figure 2-6 Use customize installation](image2)

7. Select the Coordination Manager check box for installation (Figure 2-7). Click **Next**.

![Figure 2-7 Choose the installation of Coordination Manager](image3)
8. Review the installation choices (Figure 2-8). Click **Install**.

---

**Pre-Installation Summary**

Review the following configuration before continuing:

- **Product Name:** IBM WebSphere MQ Low Latency Messaging
- **Install Folder:** /opt/ibm/1im
- **Link Folder:** /tmp/install Sir.12990/Do_Not_Install
- **Install Set:** Custom
- **Product Features:**
  - Runtime,
  - Development Kit,
  - Coordinated Cluster Time Technology Preview
- **Java VM Installation Folder:** /opt/ibm/1im/java
- **Disk Space Information (for Installation Target):**
  - Required: 224,399,772 bytes
  - Available: 22,277,554,176 bytes

PRESS <ENTER> TO CONTINUE: 

---

**Figure 2-8 Summary**

9. When the installation has been completed, a summary window is shown indicating a successful installation to the install location specified (Figure 2-9).

---

**Installing...**

[-------------------|-------------------|-------------------|-------------------|-------------------]
[-------------------|-------------------|-------------------|-------------------|-------------------]

**Installation Complete**

---

IBM WebSphere MQ Low Latency Messaging has been installed to:

- /opt/ibm/1im

PRESS <ENTER> TO EXIT THE INSTALLER: 

---

**Figure 2-9 Installation completed**

10. Click **Done**.

---

**Installation log:** If there are any errors, an installation log is created in the install location specified in step 5 on page 24.
2.1.3 Uninstall

The InstallAnywhere installer creates an uninstaller that must be used to uninstall the LLM product, if desired.

The uninstaller is created in the Uninstall directory at the install location that was specified when the product was installed. Run the uninstaller program from this location to uninstall the product. See Figure 2-10.

```
[root@localhost Uninstall]# pwd
/opt/ibm/llm/Uninstall
[root@localhost Uninstall]# ./uninstall
Preparing CONSOLI Bode Uninstallation...

IBM WebSphere MQ Low Latency Messaging (created with InstallAnywhere)

Uninstall IBM WebSphere MQ Low Latency Messaging

About to uninstall...

IBM WebSphere MQ Low Latency Messaging

This will remove features installed by InstallAnywhere. It will not remove files and folders created after the installation.

PRESS <ENTER> TO CONTINUE:
```

Figure 2-10  Uninstall script

As shown in Figure 2-11, you are able to uninstall just some features or the entire product installation.

```
Uninstall Options

ENTER THE NUMBER FOR YOUR CHOICE, OR PRESS <Enter> TO ACCEPT THE DEFAULT:

1- Completely remove all features and components.
2- Choose specific features that were installed by InstallAnywhere.

Choose one of the following options:: [root@localhost Uninstall]# 1
```

Figure 2-11  Uninstall options
2.2 Sample components

The sample components were developed in order to demonstrate the benefits of the WebSphere MQ Low Latency Messaging (WMQ-LLM) product. It shows how WMQ-LLM can satisfy the challenging messaging requirements of a financial exchange in today’s environment:

- **Extreme performance:** With extremely low microsecond latency and extremely high message volumes at rates of millions of messages per second.
- **Message delivery flexibility:** Reliable multicast message delivering with both one-to-many and many-to-many multicast messaging and point-to-point unicast messaging. It enforces a consistent order of message delivery from a number of independent data transmitters and receivers.
- **Reliable message delivery:** Reliable multicast messaging with fine-grained control of message delivery assurance.
- **High availability with fast failover:** The Reliable and Consistent Message Streaming (RCMS) component provides highly available message streaming for fault tolerance. RCMS detects a component failure and migrates the data streaming from a failed to a backup application instance.
- **Monitoring and congestion control:** WMQ-LLM provides detailed visibility into the status of transmitters, receivers and latency. It also detects bottlenecks and streamline data flow.
- **Message persistence:** Message persistence at wire speeds for message recovery and auditing.
- **High Speed message filtering:** WMQ-LLM supports fine-grained data multiplexing and efficient data segmentation.
- **High Speed Interconnects:** For the best possible latency and throughput, support for InfiniBand and 10 Gigabit Ethernet. The explosion of market data rates is exhausting the capacity of 1 GbE networks.
Figure 2-12 shows the test scenario used in this publication.

![Completed scenario for testing](image)

The application simulates the main components found in a stock exchange. The application consists of the following components:

- Two Order Submitting Clients (Client)
- Two Order Gateways (Gateway)
- Two Matching Engines (ME)
- One Market Data Publisher (Publisher)
- One Format Converter (Converter)
- One Retransmitter Service (Retransmitter)
- One Message Store (Msg Store)
- One Client Subscriber (Subscriber)
2.2.1 Order Submission Client

The Order Submission Client simulates the generation of order messages (Order, Cancel, CancelReplace,...) and sends them to the exchange. The Order Submission Client receives responses from the Order Gateway in the same message format. Because in most cases the Order Submission Client resides outside the exchange, the communication with the Order Gateway is done over a simple TCP link.

In the demonstration environment, we had two Order Submission Clients sending messages to their corresponding Order Gateway. The communication is made through RMM-TCP. Both the Order Submission Client and the Order Gateway manage the same message format. Basic information configured for the Order Submission Client is shown in Figure 2-13.

These are the principal functions performed by the Order Submission Clients in the tests:

- Generate representational orders at high message rate
- Capture round trip latency and all phases latency
- Domesticate application of RMM-TCP connection to a remote host: the exchange
2.2.2 Order Gateway

The Order Gateway receives messages from the Order Submission Client and converts them to the internal format used by the exchange. The Order Gateway then processes the message, and based on the symbol of the message, forwards the message to the appropriate Matching Engine. The Order Gateway receives responses (ActionResponse, Trade, and so on) from the Matching Engine, and the responses are converted to the correct format known by the Client and then sent to the Client.

In the demonstration environment, we had two Order Gateway receiving messages from their corresponding Order Submission Client, and the communication is made through RMM-TCP. Both Order Client and Order Gateway manage the same message format. Basic information configured for the Order Gateways is shown in Figure 2-14.

These are the principal functions performed by the Order Gateways in the tests:

- Demonstrates singleton RCMS tier
- Formats message received from Order Client to internal format and sends it to Matching Engine
- Formats message received from Machine Engine, transforms it to Client's format, and send reply
- Captures latency to/from Matching Engine
2.2.3 Matching Engine

The Matching Engine receives messages from several Order Gateways. Each message is processed according to its symbol using the appropriate order book. The outcome of the processing phase is an acknowledgment to the originating Order Gateway, as well as information about all the trades that the new message generated (if any). The trade information is sent to all the Order Gateways which originated orders involved in that trade.

In the demonstration environment, we had two Matching Engines configured on an RCMS tier. Both were receiving messages from the Order Gateway, and the communication was made through RCMS. Basic information configured for the Matching Engine is shown in Figure 2-15.

Figure 2-15  Matching Engine (ME01 and ME02)

These are the principal functions performed by the Matching Engines:

- Demonstrates high availability multi-member
- Demonstrates RCMS tier
- Demonstrates RCMS total ordering and synchrony
2.2.4 Market Data Publisher

The Market Data Publisher receives order updates, trade reports, and optionally price (quote) level messages from the Matching Engine. Then it filters those messages and publishes the latest order and book data to the receivers.

In the demonstration environment, we had one Market Data Publisher configured as a singleton RCMS tier. It is receiving messages from the Matching Engine, and the communication is made through RCMS. Basic information configured for the Market Data Publisher is shown in Figure 2-16.

![Figure 2-16 Market Data Publisher (Publisher)]

These are the principal functions performed by the Market Data Publisher:

- Filters on message type based on market data definitions
- Leverages LLM packetization to create natural payload
- Provides market data feed sequence number
2.2.5 Packet Format Converter

The Packet Format Converter receives messages from Market Data Publisher, then converts messages into generic UDP packets and publishes them to the UDP subscribers such as feed handlers. Because of the unreliable transport nature of UDP, the messages are persisted into a Message Store.

In the demonstration environment, we have one Packet Format Converter configured as a singleton RCMS tier. It is receiving messages from the Market Data Publisher, and the communication is made through RCMS. Basic information configured for the Format Converter is shown in Figure 2-17.

![Figure 2-17 Format Converter (Converter)](image)

These are the principal functions performed by the Packet Format Converter:

- Converts LLM packet/messages into a single UDP packet for Internet distribution
- The same UDP packet becomes one LLM message to be stored in Message Store
2.2.6 Message Store

The Message Store receives the messages from the Packet Converter and stores them using a policy definition, those stored messages are used to work around unrecoverable packet loss, by retrieving otherwise missed messages from the store. The Message Store provides a mechanism to get a single or a batch of stored messages identified by different filters.

In the demonstration environment, we have one Message Store configured as a singleton RCMS tier. It is storing messages it received from the Packet Converter, and the communication is made through RCMS. Basic information configured for the Message Store is shown in Figure 2-18.

![Figure 2-18 Message Store (Msg Store)](image-url)
2.2.7 Retransmission Service

The Retransmission Service retrieves messages from Message Store and sends to the requesting Client. Because UDP had no ability to retransmit lost packets, this is a way to retransmit lost packets from the Message Store to external subscribers. Late joining or restarted applications can be quickly be initialized into a given (current) state by retrieving messages from the store.

In the test environment, we have one Retransmission Service. It receives a request for retransmission from a Client subscriber. The communication with the Client subscriber is made through TCP (base we substitute with command line input for simplicity) and with the Message Store through LLMI. Basic information configured for the Retransmission Service is shown in Figure 2-19.

These are the principal functions performed by the Retransmission Service:

- Retrieves stored message (previously sent UDP packet) to support market data retransmission
- Source for delayed market data feed

![Figure 2-19 Retransmission Service (Retransmitter)]
2.2.8 Client Subscriber

The Client Subscriber receives messages from the Packet Format Converter and displays them.

In the demonstration environment, we have one Client Subscriber. It is receiving messages from the Packet Format Converter, and it also receives messages from the Retransmission Service. The communication is made through UDP. Basic information configured for the Client Subscriber is shown in Figure 2-20.

These are the principal functions performed by the Client Subscriber:

- Display market data streams from exchange.
- Display retransmitted messages from exchange.
Developing highly available matching engines with RCMS

In this chapter, we show how to extend our previous sample to create a highly available Matching Engine (ME) component using the RCMS API.

We discuss how the state of the application is transferred from the leading tier member when a tier member joins the tier. We also consider the various strategies and best practices that are available.

We demonstrate RCMS features used to keep messages in the same order, so every tier member receives exactly the same input, which guarantees the same output from the tier. We explain how to use real synchrony, virtual synchrony, and fast real synchrony, which is only available in Low Latency Messaging (LLM) 2.6.

Finally, we discuss events handling, showing what you can do when the state of the application changes or the role tier member changes.

**Terms:** In this chapter, we refer to the C APIs and structures. Similar objects and methods exist for Java/.NET.
3.1 A highly available matching engine

As you might recall from Chapter 1, “Introduction and concept” on page 1, RCMS provides all the features for high availability (HA). It uses active and semi-active replication techniques to achieve it.

This capability is especially valuable for stock exchanges or any industry where redundancy is critical. It includes situations when it is not permissible for a component to fail and affect daily operations that can cost millions in money and effort.

The idea of a highly available matching engine with RCMS, or any other application, is not really complex. As in many redundancy scenarios, more than one instance of an application are running at the same.

The instances can be active, that is, processing messages at the same time, or only one can be chosen as the primary instance and the rest of them as the backup instances. If any instance fails, operations will not be affected because the other instances take care of them.

We want to create a two-member tier for the matching engine. Each member will run in a separate server in order to show how can you achieve HA with LLM.

Although this scenario is not complex, as mentioned before, there are certain considerations that we need to keep in mind when working with the RCMS API.

3.1.1 Considerations

When using the RCMS API to develop new applications or integrate with an existing one, you need to consider characteristics such as these:

- Total order of messages
- Defining the state of the application
- Distributing the state of the application

**Total order of messages**

If the application is going to receive messages from different sources, you need to make sure that every instance of the application receives them in the same order, so each instance produces the same output. RCMS takes care of this operation, but it has different modes to achieve it, as discussed in 3.4, “Message order” on page 48.

**Defining the state of the application**

The state of the application reflects how your application has changed after processing one or more messages. This concept depends on how the application is built, but it is important because we want to be sure that at any time, all instances of the application are in the same state. More details are covered in 3.5, “State synchronization” on page 51.

Along with message order, having a tier member in the same state guarantees that they produce the same output.
Distributing the state of the application
The state of the application must be the same for all the instances of the application running at the same time, but let us consider what happens if we start a new instance later or recovering a failed tier member. There must be a mechanism for the leader instance to transfer its state to this new instance. We can see that RCMS offers a simple mechanism to achieve this, as discussed in 3.5, “State synchronization” on page 51.

3.1.2 Overview of the sample application
In this section, we describe the components and libraries that are used in the sample code in this chapter.

Order book library
This library is used to find a match based on the symbol, the number of shares, price, side amount other things when an order is received. It returns a list of affected orders and a list or trade report when a match was found.

Although this code is written in C++, it is possible to integrate it with RCMS code written in C. You only need to compile RCMS C code with the C++ compiler.

Table 3-1 shows the two classes that the order book component contains.

| Table 3-1  Classes defined for the order book component |
|-----------------|---------------------------------|
| Class           | Description                     |
| Security        | This class contains the information concerning one symbol. A security is created for each symbol that is traded. It contains an order book object which is the engine for the matching process. |
| OrderBook       | This class contains all the methods to process new orders, is responsible finding a match for matching orders. It keeps a record of every order received until the match is found for that order. |

It is really a simple matching engine and its purpose is to illustrate how the matching process takes place in this stock exchange scenario.

As we can see in the following section, the order book library is used by the Matching Engine component.

Tier library
This library includes all the functions that use the RCMS API. It is not only used by the Matching Engine component but all the other components that are RCMS singletons.

Table 3-2 shows all the functions included in this library.

| Table 3-2  Functions in Tier library |
|-----------------|---------------------------------|
| Function        | Description                     |
| create_rcms_tier() | This functions initializes a tier member and all the parameters needed. It calls parse_configuration_file() to parse the configuration file and setup each parameter. |
| parse_configuration_file() | This function does all the parsing of the configuration file for each tier member. |
| create_rmcs_tier_tx_topic() | This function creates the RCMS transmitter topic. |
All the functions in this library use the RCMS C API.

Matching Engine component
The Matching Engine (ME) component is responsible for receiving orders from the Gateway. It tries to match each order with the previous received orders stored in the order book. After this process is completed, it sends the list of affected orders to both the Publisher and Gateway. If a match is found, a trade report for each matched order is sent to the Publisher.

The Matching Engine employs the order book and tier library for all its operations.

ME uses a global map to keep all the securities in memory, as we see later, this is basically the state of this application.

Receiving orders
ME uses on_packet() RCSM callback, so multiple RMM messages are received at once. Every RMM message is processed individually and in sequence (no parallel threads to keep the sample code simple).

The RMM message is first casted to an ME message (application message type) and then validated. If the message is an AddOrder message, the matching engine function is called as shown in Example 3-1.

Example 3-1  Casting the RMM message and calling the matching engine

```c
void process_message(rmmRxMessage *message) {
    ...
    me_message = (T_ME_MSG*) message->msg_buf;
    setupMETimeStamp(me_message, TS_GW_MEC_IN);
    ...
    rc = call_matching_engine(me_message);
}
```

Inside the call_matching_engine() function, ME looks for the security of the symbol in the global map of securities. If it does not exist, it creates a new object and add it to the global map (Example 3-2).

Example 3-2  Looking for an existing Security object

```c
T_SymbolSecurityMap::iterator itSec = allSecurities.find(symbol);
if (itSec == allSecurities.end()) {
    pSec = new Security(symbol);
    allSecurities.insert(T_SymbolSecurityPair(symbol, pSec));
} else {
    pSec = (Security*) itSec->second;
}
```
Then the order book library is used to find a match for this order. Because the order book object of the security expects a Book Order message, the ME message is transformed into this type. After doing so, the new order is added to the Security and starts to look for a match based on the previous orders received and the values of the order.

After the matching process is completed, the list of affected orders is cleaned up to remove any filled or canceled orders (Example 3-3).

**Example 3-3  Finding a match for the order**

```c
...  order = transformMEMessage2MBookOrder(message);  pSec->SME.NewOrder(&order, &affectOrders, &trades); ...  pSec->SME.RemoveFilledOrders(&affectOrders); ...
```

**Publishing confirmations and trade reports**

After this matching process is completed, for every order included in the affected order list, an UpdateOrder ME message is sent to the Publisher and the Gateway. Notice that a transformation between is needed to create this reply.

If a match or matches were found in any order in the securities, a trade report is sent to the Publisher. In this case, the message is a T_TradeReport and not an ME message (Example 3-4).

**Example 3-4  Replying the Gateway and Publisher for each order received**

```c
...  while (itO != affectOrders.end()) {
      pOrder = (T_BookOrder*) (*itO);  
      output_me_msg = transformBookOrder2UpdateOrder(pOrder);  
      send_reply_gateway(&output_me_msg);  
      send_to_publisher(&output_me_msg);  
      itO++;
  } ...

  while (itT != trades.end()) {
    pTrade = (T_TradeReport*) (*itT);  
    send_tradereport_to_publisher(pTrade);  
    itT++;
  }

  affectOrders.clear();  
  trades.clear();
```

Just before finishing, the affectOrders and trades lists are clean because they are global variables in this applications.
3.2 Setting up an RCMS tier

The sample code showed in Chapter 2, “The demonstration environment” on page 9 is already using the RCMS API. The only difference is that every component (Gateway, Matching Engine, publisher, and UDP Converter) is configured as a member of a singleton tier.

A singleton tier provides no HA. The main place where we expect to find a singleton tier is in the end clients that are not made highly available themselves, but need to receive data from servers that are sending the data over tier topic transmitters.

In this section, we discuss the changes that are needed to be done in the Matching Engine component to become a fully HA component. The idea is to create a two-member tier. We want each member to run as a separate server. We refer to this tier as the ME tier.

Figure 3-1 shows the components,

![HA components](image)

For simplicity in the paper, we have only chosen the Matching Engine component to illustrate the procedure. The other components can also be modified to become fully HA tiers.

3.2.1 RCMS configuration file

A RCMS tier can be configured to work in several configurations. The configuration is set in the rmmTierParameters structure that is used when creating the tier.

A good practice is to create a configuration file so every parameter that needs to be set up for the tier is there and any change can be done easily at any time with no code modifications.

It is important to remark that there is not a predefined configuration file layout for RCMS. You are free to define your own, from stanza files (such as the one we are using with the sample) to XML files. This offers you a flexible way to integrate RCMS with already existing applications.

At the end you only need to code the way the configuration file is parsed. In our sample code that work is done by the parse_configuration_file_mul() function in the tier library, so you can take a look to learn from this example.

Two configuration files are needed, one for each member in the ME tier.
3.2.2 LLM parameters for RCMS

Because this is not a singleton tier, it must initialize intra-tier communication parameters and HAC configuration parameters before the tier is initialized.

The required parameters for each member are shown in Table 3-3. Every parameter is a public attribute of the rmmTierParameters struct (rmmCApi.h file).

Table 3-3 TierInfo stanza parameters for RCMS

<table>
<thead>
<tr>
<th>Parameter in config file</th>
<th>rmmTierParameters attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>member_name</td>
<td>The name of this member in the tier; it must be unique within the same tier.</td>
</tr>
<tr>
<td>Singleton</td>
<td>is_singleton</td>
<td>Setting up this value to zero means the member belongs to a non-singleton tier and the intra-tier communication parameters and HAC configuration parameters need to be setup before the tier is initialized.</td>
</tr>
<tr>
<td>memberCount</td>
<td>nmembers</td>
<td>Indicates the number of the members in the tier. Each tier member must know the address and port for all the other members in the tier.</td>
</tr>
<tr>
<td>member</td>
<td>member_name</td>
<td>The list of each tier members with its address and port. This address and port are for each tier member to accept inter-tier communication and being in synchrony with the other tier members.</td>
</tr>
</tbody>
</table>

For Control Traffic parameters (ctrl_address, ctrl_topic_name, ctrl_port), we use the default values, you can refer to the API documentation for C included with LLM for these values.

For High Availability Coordinator (HAC) parameters also, we are using the default values, except for HAC instance name, as shown in Table 3-4.

Table 3-4 HAC parameters in tier configuration file

<table>
<thead>
<tr>
<th>Parameter in config file</th>
<th>rmmHacParameters attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HACTierName</td>
<td>hac_instance_name</td>
<td>The name must be the same for all the members of the same tier. Note that if you want to run multiple RCMS tier instances in the same process, you must set a different name for each HAC instance.</td>
</tr>
<tr>
<td>hacStartupTimeout</td>
<td>hac_startup_timeout _milli</td>
<td>Time in milliseconds to wait for other members on startup. The minimal value is 5000. Default value 60000. This is the time a member is going to wait trying to connect to each tier member. After this time, the tier member gives up and marks the other tier member as down. You need to restart the other tier member.</td>
</tr>
</tbody>
</table>
The config file for first member in the ME tier looks as shown in Example 3-5,

Example 3-5  TierInfo stanza fragment in the tier member 1 configuration file

```
[TierInfo]
Name=ME1
#ctrlAddress
#ctrlPort
#ctrlTopicName
RealSynchrony=0
Singleton=0
TotalOrder=1
#syncPeriodMicro=1000
#maxSyncAttempts
#syncIntervalMilli
#recoveryTimeoutMilli
hactiername=ME_HAC
hacStartupTimeout=20000
memberCount=2
member=ME1@10.0.0.2:10101
member=ME2@10.0.0.3:10102
```

For the second member, the values are the same except for the name of the tier member (remember that tier member name must be unique in the tier).

### 3.2.3 RCMS in C/C++

The sample code in this paper is written in Java. As mentioned before, in our sample application, we are going to use C/C++ to illustrate how to use the RCMS API with this language.

We encourage you to compare this with the sample code in this paper to see how it is done in Java.

**Initializing a tier member**

Each tier member is initialized when create_rcms_tier() is called in the sample code. The same as in Java, to create a new tier member, you need to initialize an rmmTierParameters variable (tier_config in the sample) and pass it out to rmmTierInit() as shown in Example 3-6.

Example 3-6  Initializing the tier member

```
... if (rmmInitStructureParameters(RMM_SID_TIER_PARAMETERS, tier_config,
       RMMCAPI_VERSION, &expCode) != RMM_SUCCESS) {

... // more configurations here

printf("Creating RCMS tier...\n");
if (rmmTierInit(tier, tier_config, &expCode) != RMM_SUCCESS) {
    printf("Failed to initialize RCMS tier (Explanation code = %d)\n",
            expCode);
    rc = 4;
}
... 
```
Registering tier members
To register every member for the tier. You specify the number of members in the tier. For each member, just populate the rmmTierMember struct and set up the name, address, and port as shown in Example 3-7.

Example 3-7   Registering tier members in C

```c
... // before rmmTierInit is called
    tier_config->tier_members = (rmmTierMember *) malloc(
        sizeof(rmmTierMember) * tier_config->nmembers);
...

    strncpy(
        tier_config->tier_members[iTierIdx].member_name,
        pStart,
        sizeof(tier_config->tier_members[iTierIdx].member_name));
...

    strncpy(
        tier_config->tier_members[iTierIdx].tier_address,
        pStart,
        sizeof(tier_config->tier_members[iTierIdx].tier_address));
...

    tier_config->tier_members[iTierIdx].tier_port = atoi(pStart);
...
```

You can find this code in the parse_configuration_file() function in the tier library.

Message callbacks
When we use RCMS, as when using RMM, it is only possible to use on_packet() or on_message() callbacks; no other mode is supported.

In Example 3-8, we are using the on_packet() callback which means several messages will be received and each message is processed individually and in sequence.

Example 3-8   on_packet callback implementation in ME

```c
void on_receive_packet(rmmPacket *packet, void *user) {
    int i = 0;
    for (i = 0; i < packet->num_msgs; i++) {
        process_message(&packet->msgs_info[i]);
    }
}
...
```

Because it is possible to have more than one topic for each transmitter or receiver tier instance, the callback is set up when the topic is created (function create_rmcs_tier_rx_topic()).

Registering state callbacks
Every time a new tier member joins the tier, LLM calls the rmm_get_state_object_t get_state_object API on the leading tier member, and the rmm_set_state_object_t set_state_object API on the new tier member side, to initiate the transfer of the state of the application.
You need to implement both callbacks and set up them before the tier is initialized (see Example 3-9).

Example 3-9  State callback defined in ME and library tier

```c
int set_state_object_me(const rmmAppState *state_object, void *user) {
    ... // if no error has occurred
    return RMM_SUCCESS;
}

int get_state_object_me(rmmAppState *state_object, void *user,
                       char* TierMemberName) {
    Security* pSec = NULL;
    ... // if no error has occurred
    return RMM_SUCCESS;
}
...
```

We describe in more detail how this process takes place in 3.5, “State synchronization” on page 51.

Role member callback
Each tier member can be in one of four different states: UNSYNC, MEMBER, LEADER, or ERROR. Every time a tier member changes its state, LLM calls the rmm_tier_role_changed_t role_changed API.

In 3.3, “Events handling” on page 46, we discuss more about this callback.

3.3 Events handling

LLM supports different callbacks that help to notify the application when something in LLM has occurred or changed. We already talked about a few of them before, such as message callback or state callbacks (see 3.2.3, “RCMS in C/C++” on page 44).

In this section, we cover two of these callbacks: change of member role callback, and state synchronization callback.

3.3.1 Change of member role callback

As mentioned before, a tier member always has a state/role and it can change several times along the tier member lifetime.

Whenever this change takes place, LLM invokes the rmm_tier_role_changed_t role_changed() API in order to notify the application about it.

This callback is not necessary for the tier to operate, so it is null by default. In our example, we implemented it as shown in Example 3-10. As usual with all the callbacks, you need to register this function in the role_changed parameter before the tier member is initialized.
Example 3-10  Capturing changes in member role

```c
void role_changed(rmmTierRoleInfo *role_info, void *tier_user) {
    ...
    switch (role_info->new_role) {
        ...
    }
    switch (role_info->old_role) {
        ...
    }
    ...
    printf("--> Active members = %d, Sync members = %d, ",
            role_info->num_active_members, role_info->num_sync_members);
    
    if (role_info->tier_leader != NULL)
        printf("--> Tier leader is %s.\n", role_info->tier_leader->member_name);
}

int create_rcms_tier(rmmTier* tier, rm ... ) {
    // just before initializing the tier member
    tier_config->role_changed = role_changed;
    ...
```

A common use for this callback is for monitoring the state of the tier member. For instance, it allows you to detect when an error has occurred (when the state is ERROR).

Another usage can be to hold any message processing while the state of the application is completely synchronized when a fail tier member is recovering. While the state does not change from UNSYCN to MEMBER or ROLE, the tier member is not full active and is not processing any message.

If you are interested in measuring how long it takes to transfer the whole state of the application from one tier member to another, you can use this callback to achieve that.

### 3.3.2 State synchronization callbacks

There are two other callbacks that are useful in LLM. The first, rmm_sync_completed_t sync_completed(), is invoked when the process synchronization of the state of the application is completed with no problem, that is, when RMM_SUCCESS is returned. This callback is called on the side of the new tier member that is recovering from a failure or joining the tier for the first time.

This allows you to notify other running components when the tier member is active and ready to start processing messages.

The other one is rmm_sync_failed_t sync_failed(). If anything but RMM_SUCCESS is returned by the get_state_object() or set_state_pobject() callbacks during the synchronization of the state of the application, this callback is invoked.

This last callback is useful because it can warn other components that the transfer of the state of the application was not successful and some measures need to be taken.

In our sample, we just implemented these two callback to print out a messages whenever any of them is invoked.
Example 3-11 shows how these two callbacks are implemented in our sample. You will find this code in the util library (util.c/util.h).

**Example 3-11  State synchronization callbacks**

```c
void sync_completed(void *user) {
  ...
}

void sync_failed(RMM_HAC_ERROR_t reason, void *user) {
  ...
}

int create_rcms_tier(rmmTier* tier, rm ...) {
  // before initializing the tier member
  tier_config->sync_listener.sync_completed = sync_completed;
  tier_config->sync_listener.sync_failed = sync_failed;
  ...
}
```

It is important to notice that these two callbacks need to be implemented, otherwise the tier member will not start.

### 3.4 Message order

To illustrate why message order is critical when you have more than one instance of the application running at the same time, let us start with the following example.

Suppose we have a two-member tier up and running and we have two gateways sending orders to it. Each gateway receives orders from different clients as shown in Figure 3-2.

![Figure 3-2  Scenario with two gateways and ME tier](image)

The order book on each tier member has 100 shares for IBM available to sell. At this time, both tier members have the same state.

Then Client 1 sends an order to buy 70 shares and Client 2 sends an order for 150 shares (let us keep our example simple and suppose that each order has the same probability to get a match).

Because all the components are running on different computers, it is possible that messages coming from each gateway might be received in a different order by each tier member.
Let us say that the order from Client 1 gets received before the order from Client 2 in Member 1 and the opposite way for Member 2, just as shown in Figure 3-3.

As you can see, there is no way that the result coming out from each member can be the same. For Member 1, the order 1 was fulfilled while the other one is partially fulfilled. On the other hand, there is a completely different result; order 1 was partially fulfilled while there is no match for order 2.

Now the question is, what result is valid and which is going to be sent to the publisher. If both members are allowed to send their result to the publisher, the final result is a contradiction that can cause serious problems because both results are different.

The same happens if only one member, the primary, is allowed to send the results to the publisher. Which result do you say is the correct one?

To make things more complicated, what happens if a failing tier member is rejoining the tier. Which state of the application does this member have to receive?

It is clear that this lack of synchrony between each tier member due a different order in the messages creates an unnecessary and complex dilemma.

The solution is to guarantee that the same input is received by each tier member. Receiving the messages in the same order means that the same output must be produced. RCMS offers several modes to achieve this.

### 3.4.1 Total order

A total order in RCMS is only needed when you have more than one input source. If you have a single input source, then RMM is enough to handle the message order.

By default, the total order is enabled. If by any reason you do not want to use it, just set is_total_order to zero in the rmmTierParameters struct before the tier member is initialized.

In our sample, a total order is needed, because we are going to use two gateways.
3.4.2 Modes of synchrony

LLM supports different modes to achieve message order and avoid the lost of synchrony between tier members.

These are the modes:

- Virtual synchrony
- Real synchrony
- Fast real synchrony (available in LLM 2.6)

Each one offers a different balance between lower latency and higher resiliency to failures. Let us review each one in the following sections.

Virtual synchrony
Working with a virtual synchrony tier provides the lowest possible latency and ensures that the application is protected against any single failure. However, virtual synchrony cannot completely ensure full recovery from multiple simultaneous failures.

Real synchrony
Real synchrony provides protection against all failures (except complete tier failure) with the price of an increase in latency. The additional latency introduced by working with real synchrony is roughly the maximal round-trip time between any two members in the tier.

Fast real synchrony
By default, RCMS sends control information, including total ordering information and other intra-tier messages, between tier members by using the group communication services of DCS. This is done for both virtual synchrony and real synchrony tiers. For a virtual synchrony tier, the end-to-end latency is not affected by DCS performance, but with a real synchrony tier, the latency is strongly affected by the round trip latency over the DCS service. The reason is that with real synchrony, the primary has to wait for an acknowledgment to the ordering information it sends from the backup tier members before the data can be delivered to the application.

To improve the latency of a real synchrony tier, RCMS allows an application to override the default behavior and send the ordering information over a dedicated fast control channel. The fast control channel is implemented using a regular RMM topic that provides the lowest latency possible.

In order to support the option of using a fast control channel, a specific unicast port, called real_synchrony_fc_port, was added to the rmmTierParameters structure. This port is used for the fast control communication between the tier members.

3.4.3 Testing scenarios

By default, the mode that is enabled is virtual synchrony. We test two different scenarios in the following chapters to show what are the advantages and disadvantages of the following comparisons:

- Real synchrony versus fast synchrony
- Fast synchrony versus fast real synchrony
3.5 State synchronization

As mentioned before, knowing exactly how the state of the application is defined is fundamental when you work with RCMS.

You might remember that all the tier members need to be in synchrony. What happens when a tier member fails and after a period of time it rejoins the tier?

For a stock exchange application, probably in that period of time, hundred or thousands of orders have been processed. How can the failing tier member be in synchrony or same state as the other tier members?

There are two possible options for this tier member:

- Reprocess all the orders from the very beginning, that is, from the first order received in the day
- Make a copy of the current state of the leading tier member.

The first option usually is not optimal because it requires too much time (it depends, of course, on the rate of messages/orders received by the tier by the time the failing tier member restarts).

The second option is more viable, simple, and seems logical. Actually there is a hidden detail, you need to stop processing any message while you are copying the state of the leading tier member. Keep in mind this idea as we address this topic again in 3.5.2, “How LLM state synchronization works” on page 52.

So in order to implement the second point, we just need to define very precisely, from the point of view of our application, what is the state of the application at a given time.

3.5.1 State definition

So what is the state of the application? Well, that answer depends on your application. You need to ask yourself what changes occur in your application after a message is processed.

For stock exchange, the answer is simple: It is the order book.

In our sample, each tier member has its own order book in memory. Every time an order is received, the order book changes if a match is found.

We use a global map to keep a list of all the securities that hold the order book for each symbol that has been received by a tier member.

This global map represents at any time the state of our tier member, the state of this application.

For your application, you need to think what will be the equivalent. The key question to ask is, what entity in my application changes every time a message is processed?

After you have a clear idea of how is defined in your application the state, we need to understand what is required to copy it to another tier member.
3.5.2 How LLM state synchronization works

As you might remember from 3.2.3, “RCMS in C/C++” on page 44, every time a failing tier member rejoins the tier (or a new one), a callback is invoked in the leading tier member side to notify it that a copy of the current state is required.

In the failing tier member, another callback is invoked when the state information is received from the leading tier member.

The state is transferred in a serialized form inside an attribute of the rmmAppState struct. The serialization can be as simple or complicated as you want it, you just need somehow to convert the state of your application to a string or buffer.

The attributes for the rmmAppState struct are shown in Table 3-5.

Table 3-5  rmmAppState struct

<table>
<thead>
<tr>
<th>Public attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>When used in get_state, this is the buffer to which the application must copy its state data.</td>
</tr>
<tr>
<td>buffer_length</td>
<td>Length in bytes of the buffer RMM provides to the application.</td>
</tr>
<tr>
<td>data_length</td>
<td>Number of valid bytes held in the buffer.</td>
</tr>
<tr>
<td>state_sqn</td>
<td>Sequence number of the state which is currently handled.</td>
</tr>
<tr>
<td>fragment_sqn</td>
<td>Sequence number of the fragment in the current state.</td>
</tr>
<tr>
<td>is_last_frag</td>
<td>Indicates that this fragment of the data is the last one.</td>
</tr>
<tr>
<td>synchronize_call</td>
<td>Input/output parameter.</td>
</tr>
<tr>
<td>next_call_milli</td>
<td>Time in milliseconds that the application wants RMM to wait before invoking the next call to get_state_object.</td>
</tr>
<tr>
<td>rsrv</td>
<td>Reserved attribute.</td>
</tr>
</tbody>
</table>

Sending out the state

On the leading tier member side, before any message is sent, you need to lock the state of the leading tier member. The lock is necessary in order to prevent the leading tier member from continuing to process messages. Otherwise, the state will be constantly changing as the tier is still receiving messages.
You do this by setting up the two parameters shown in Example 3-12.

**Example 3-12  First message sent by get_state_object() callback in the sample code**

```cpp
... if (!state_object->synchronize_call) {
    cout << "synchronization : first part" << endl << flush;

    // set state object values
    state_object->synchronize_call = TRUE;
    state_object->data_length = 0;
    counter = 0;
} else {
...
}
return RMM_SUCCESS;
```

You can see that no information is sent, state_object->data_length = 0. You only need to indicate the leading tier member to stop processing message, state_object->synchronize_call = TRUE, while the whole synchronization process for the state lasts.

Also notice that RMM_SUCCESS must be returned if no error has occurred.

After this first part, you can start sending the serialized state string.

What happens if this string is too big? No problem, you can split apart into small chunks or fragments and send one at the time. Just use the attribute fragment_sqn to indicate the number of the chunk that is being sent (Example 3-13).

**Example 3-13  Sending a fragment of the serialized state string**

```cpp
... state_object->fragment_sqn = counter++;
memcpy(state_object->buffer, g_app_state_XferBuffer,
    g_app_state_XferBufferLen);
state_object->data_length = g_app_state_XferBufferLen;
...
return RMM_SUCCESS;
...```

You can continue sending fragments until all the information is sent.

To indicate that the synchronization process is finished, just set TRUE to the attribute is_last_frag of armmAppState, as shown in Example 3-14.

**Example 3-14  Indicating that the synchronization process is over**

```cpp
... state_object->data_length = 0;
state_object->is_last_frag = TRUE;
return RMM_SUCCESS;
...```
After the last fragment is sent, the process is completed and the leading tier member starts to processing messages again.

The whole process is illustrated in Figure 3-4.

In our sample application, the first fragment sent contains only an empty list of all securities stored in the global map in the leading tier member.

The serialized form, if this empty list is just a pipe separated list with all the names of the symbol, is shown in Example 3-15.

**Example 3-15  Example of the serialized list of securities**

| IBM | SYM2 | SYM3 | ...

Then, we use a fragment for all the orders corresponding to a symbol. The serialized string is similar to the one shown in Example 3-15.

**Example 3-16  Example of the orders corresponding to a symbol**

| IBM | Order 1 information | Order 2 information | ...

This process is continued until all the orders has been sent to the new member.

**Receiving the state**

On the failing or new tier member side, the process is much simple. Every time the leading tier member sends you a piece information, the set_state_object() is called.

You just need to use the information in the fields to deserialize the state of the application and then recreate it in this member.

In our application sample, when the first fragment is received, we deserialize the pipe delimited string and recreate the list of empty securities.

On the following fragments, the order list for each symbol is built.
3.5.3 Considerations for state synchronization

There are a few considerations that you need to keep in mind when working with state synchronization.

For instance, when you use a synchronize call (sync|onize_{call} = TRUE) you are only telling the leading tier member to stop processing messages but this does not stop it from receiving them.

So you want to be sure that the synchronization process for the state is as fast as possible. If it takes too much time, you are going to lose messages, depending on the size of the tier receiver buffer.

Another consideration is that LLM already has an allocated buffer to hold the serialized state, so we do not need to allocate it every time the get_state_object() is called. As mentioned in Table 3-5, “rmmAppState struct” on page 52, the attribute buffer_length says how big is this buffer. Keep in mind that your application must not write more than the buffer_length bytes into the buffer.

3.5.4 Best practices

Here is a list of best practices to follow when working with state synchronization:

- Have a clear and precise idea of what the state of your application means in your application context. If you are not sure what is the state of your application, it is highly probable that you are not going to transfer it correctly to the other tier member.
- Keep the memory footprint for the state of the application as small as possible, in order to minimize the time employed to serialize or deserialize the state.
- Make the whole synchronization process as short as possible in time, so you will not lose messages.
- Transfer the state in fragments, as many as you need, but keep in mind the timing restriction.
- Send only the basic elements to build the state in the receiver tier member side. That will minimize the amount of information that you need to send along the network between the two members and also the amount of time required.

3.6 Filtering

In the scenario with two gateways running and sending orders to the ME tier, we need to consider that the reply from ME tier (UpdateOrder message) can be received by the two gateways.

That is caused because we are using a single topic for communication between each gateway and ME tier. And both gateways are subscribed to the same multicast group, so they are pulling out the same message when the tier replies.
If we want the reply to be received only by the sending gateway, we have two options:

- Use a different topic for each gateway.
- Use some kind of filtering at the transport level on each gateway.

The first option requires that every time a new gateway is added to the system, we are forced to update the configuration file for each ME tier member to take into account the new topic.

The second option requires no modification to the config file because we can have LLM do all the filtering for us. This not only means less programming effort but more efficiency because the filtering is done at the transport level instead of the application level. No code or configuration file needs to be modified when you add a new gateway to the system.

There are a number of ways that messages can be filtered with LLM. The two most common ways of filtering messages on a single stream are through the use of message properties or turbo flow labels.

In our sample application we decided to use turbo flow labels because they do the work needed for our purposes.

### 3.6.1 Turbo flow filtering

Turbo flow filters can be either integers (labels) or bitmaps. Turbo flow message filtering provides the most efficient mechanism for filtering messages at the LLM messaging layer.

Using the turbo flow label requires the following tasks:

- Choosing the desired turbo flow mode before the topic is created.
- Setting the label before the RMM message is transmitted (on the transmitter side)
- Setting which labels are going to be accepted (on the receiver side)

Because labels are simple integers, we can use the ID each gateway for the label.

Suppose gateway 1 sends an order to the ME tier and a match is found for this order. The ME tier is going to reply with a message with turbo flow label set to 1. Because each gateway is configured to accept only messages whose turbo flow label matches with the gateway ID, only gateway 1 will receive the reply. For gateway 2, the message will be filtered by LLM and discarded.

This situation is illustrated in Figure 3-5.
Setting turbo flow mode on the ME side
To use the label mode, we need to set the attribute turbo_flow_mode of rmmTxTopicParameters before we create the transmitter topic to RMM_TF_MODE_LABEL (Example 3-17).

Example 3-17  Setting up turbo flow mode in the Matching Engine

```c
... for (i = 0; i < TX_TOPICS_MAX; i++) {
    tx_topics[i].parameters.turbo_flow_mode = RMM_TF_MODE_LABEL;
    rc = create_rmcs_tier_tx_topic(&tier, &(tx_topics[i].topic),
                                 &(tx_topics[i].parameters));
    if (rc) {
        printf("Failed to create RCMS Rx Topic.\n");
        exit(rc);
    }
}
...```

Setting turbo flow label on the ME side
This is just a matter of choosing the proper value in the msg_label attribute for the message before LLM sends it (Example 3-18).

Example 3-18  Setting turbo flow label for a RMM message

```c
int send_llm_message(int index, void *message, int length, int label) {
    ...
    tx_topics[index].tx_msg.msg_len = length;
    memcpy(tx_topics[index].tx_msg.msg_buf, message, length);
    tx_topics[index].tx_msg.msg_label = label;
    rc = rmmTxSubmitMessage(&(tx_topics[index].topic),
                            &(tx_topics[index].tx_msg), &expCode);
    ...
}
```

Example 3-18 shows our generic function for sending RMM messages. In our sample we invoke it as shown in Example 3-19.

Example 3-19  Replying to the gateway using gateway ID as turbo flow label

```c
void send_reply_gateway(T_ME_MSG* message) {
    int rc = 0;
    static int reply_gw = 0;

    setupMETimeStamp(message, TS_ME_GW_OUT);
    rc = send_llm_message(1, message, message->Len, message->gatewayId);
    ...
}```
Setting turbo flow mode on the Gateway side
The same as in the case of the ME Tier, to enable turbo flow mode, you just need to set the turbo_flow_mode as shown in Example 3-20.

Example 3-20 Setting turbo flow mode in the gateway side

```c
... 
publisher_rx_topic_params.turbo_flow_mode = RMM_TF_MODE_LABEL;
rc = create_rmcs_tier_rx_topic(&publisher_tier,
    &publisher_topic_r,
    &publisher_rx_topic_params,
    publisher_rx_MCG,
    &publisher_on_receive_packet);
... 
```

Setting turbo flow label on the Gateway side
After creating the topic, just use rmmRxAddAcceptedFlow API to specify what label the receiver must expect messages for (Example 3-21).

Example 3-21 Adding turbo flow label for gateway ID

```c
...
    rmmRxAddAcceptedFlow(&publisher_topic_r, gatewayID, &rc);
    if (rc) {
        printf("Failed to register turbo flow label for RCMS Rx Topic.\n");
        exit(2);
    }
...
```

3.6.2 Conclusions
By filtering the message using LLM, we not only save effort on coding, but we have reduced the network traffic by avoiding delivery of non-request messages to the gateway singletons.

3.7 Summary
In this chapter, we have discussed in detail many of the features that are available in RCMS. Although all the code is written in C/C++, it will not require a considerable effort to translate the samples to Java or .NET APIs.

The most important thing is this chapter is to illustrate all the fine details that you need to consider when working with highly available components, such as the definition of the state of the application, state synchronization.

We also talked about the important of keeping the synchrony on every tier member and what modes are available to keep the same receiving message order on each tier member.
Persistenc

In this chapter, we discuss the Coordination Manager Server, which is a component of WebSphere MQ Low Latency Messaging, and which provides a complementary message persistence solution to store messages and events into the data space. The Coordination Manager Server, used in conjunction with the client interface libraries (LLMI), provides the ability to store, retrieve, and replay messages from persistent storage.

The information in this chapter is extracted from the LLM Information Center, at the following Web page:


The ability to store and retrieve messages from the disk provides several major benefits. Applications can utilize the storage to work around unrecoverable packet loss, by retrieving otherwise missed messages from the Coordination Manager Server. Late joining/restarted applications can be quickly initialized into a given (current) state (provided that new messages are not coming in as quickly as the Coordination Manager Server can send old ones out). This is achieved through a replay of the messages from the Coordination Manager Server and thus minimizes any impact onto the actual transmitter originating these messages.

Important: The Coordination Manager Server is installed during the Low Latency Messaging installation:

- Ensure that you select the Coordination Manager Server check box on the Choose Product Features panel of the installer. The Coordination Manager Client is installed with the Development Kit.
- Ensure that you select the Development Kit check box on the Choose Product Features panel of the installer.
4.1 Overview

In this chapter, we demonstrate an infrastructure about the trade and book data publish and message persistence that is commonly seen in the stock exchange industry. The infrastructure diagram is described in Figure 4-1.

A few components are designed here to simulate the themes to perform the trade and book data publish and message persistence.

You can easily reuse the applications under your own exchange environment.

![Market Data Distribution Diagram](image)

**Figure 4-1  Market data distribution**

### 4.1.1 Market Data Publisher

Market Data Publisher will get order updates, trade reports, and optionally price (quote) level messages. The primary requirement for Market Data Publisher is to publish the latest order and book data to the receivers who subscribe topic PUB_TO_WORLD through RMM API:

1. Create an RCMS tier to gather data over ME_TO_PUB topic from Match Engine Tier. Here we have only a single member with a publish tier. You can define more than one member in the publish tier in your exchange production environment.

2. Filter the irrelative messages by the msg_type defined in application. We suggest the LLM message filter features, such as message properties and turbo flow, which are adapted into the stock exchange industry because of high performance in message filtering.

3. Finally, the valid messages will be published to the converter through RCMS over pub_to_world topic for distribution to subscribers outside.
4.1.2 Packet format converter

The primary requirement for packet format converter is to convert RMM messages into generic TCP or UDP packets and then publish them to the UDP subscribers such as feed handles. Because of unreliable transportation in UDP, we need to persist the UDP packets into storage such as an LLM message store each time we send out them to subscribers:

1. Subscribe to the PUB_TO_WORLD topic through RCMS to receive data from Publisher.
2. Convert order and book data into UDP packet to publish to the external market data subscriber.
3. UDP does not guarantee that a datagram will be sent to the destination successfully. We need to feed order and book data into the Coordination Manager through the CVT_TO_MSGSTORE topic because of UDP unreliable transportation.

4.1.3 Retransmit

Because UDP had no ability to retransmit lost packets, we need to implement a retransmitter to retrieve data from Coordination Manager to retransmit the loss packets to external subscriber. It will perform the following tasks:

1. Map the input of the application sequence into the timestamp that data is persisted into Coordination Manager.
2. Retrieve the messages from Coordination Manager.
3. Filter the messages whose sequence number are out of range.
4. Distribute the messages to those subscribers, such as feed handles.

Sequence number: LLM also defines a sequence number that can be used by our application. See the definition of RmmRxMessage. In this paper, we mainly talk about the sequence number defined in our application.
4.2 Configuring an LLM Coordination Manager for order and trade persistence

The Low Latency Coordination Manager (LLM-CM) Server runs as a daemon (LLMD) to do the storage, and provides an interface library (LLMI) to access stored messages. LLMD can also be used to do message bridging. The LLMD process takes object configuration information as input, and sends monitoring and logging information as output. It creates a message store on disk. Messages are sent between the LLMD and LLMI using TCP sockets. Low Latency Messaging multicast and Low Latency Messaging unicast messages are received and transmitted by the Low Latency Messaging libraries (librmm and librum).

The LLMD is configured by defining a set of Low Latency Messaging objects. For descriptions of the meaning of Low Latency Messaging configuration items, see Table 4-1.

**Services:** The WebSphere MQ Low Latency Messaging Coordination Manager provides a set of services that extend the core messaging features in RMM, RCMS, and RUMIn log messages, Coordination Manager is also referred to as LLM-CM.

4.2.1 LLMD configuration description

The LLMD configuration file is a flat file consisting of init commands and stanzas. Blank lines, and lines starting with a # or * are ignored. Each stanza is introduced by a stanza name that is enclosed in brackets, which also ends the previous stanza.

A stanza can be ended using the [End] stanza introducer. The stanza names are listed in Table 4-1.

<table>
<thead>
<tr>
<th>Stanza name</th>
<th>XML name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[LLM]</td>
<td>LLMInstance</td>
<td>Main configuration for LLM Service</td>
</tr>
<tr>
<td>[Receive]</td>
<td>LLMReceive</td>
<td>Receive configuration for LLM Instance</td>
</tr>
<tr>
<td>[Transmit]</td>
<td>LLMTransmit</td>
<td>Transmit configuration for LLM Instance</td>
</tr>
<tr>
<td>[Tier]</td>
<td>LLMTier</td>
<td>Tier group configuration for LLM Instance</td>
</tr>
<tr>
<td>[Unicast]</td>
<td>LLMUnicast</td>
<td>Reliable Unicast configuration for LMM Instance</td>
</tr>
<tr>
<td>[Store]</td>
<td>LLMStore</td>
<td>Message store object</td>
</tr>
<tr>
<td>[Policy]</td>
<td>LLMPolicy</td>
<td>Policy definition</td>
</tr>
<tr>
<td>[Route]</td>
<td>LLMRoute</td>
<td>Routing definition</td>
</tr>
<tr>
<td>[End]</td>
<td></td>
<td>End processing for an object</td>
</tr>
</tbody>
</table>

Example 4-1 shows a sample of configuration entries that precede the first stanza in our test. As can be seen from this example, within each stanza, each line contains a keyword=value pair. There can be spaces before the keyword, and around the equal sign. If the keyword represents a string value, then it will consist of all characters after the first non-blank on the line. For non-string values (including names), text after the first blank is considered a comment. See the foregoing tables for the keyword names.
An alternative implementation of the configuration is to use the XML configuration. In this mode, the objects are element names are those shown in the XML Name column above. The name attribute of these elements gives the name of the object, and the domain attribute gives the domain name. Any other element without content is ignored. Elements with content are considered configuration items. The keyword is the element name, and the content is the value. Commands are not allowed in the XML format. The XML form is specified by doing an `xconfig` command in a configuration file.

The XML data is interpreted as being in Unicode UTF-8 unless there is a byte order mark (BOM) for another Unicode encoding, or an encoding statement in the XML processing instruction. The set of supported encodings are platform dependent, but on most platforms the only supported encodings are UTF-8 and iso8859-1.

### 4.2.2 LLMD configuration example file

By default, there is an LLMD sample configuration called llmd.cfg in the LLM_installation_directory/samples/config directory that can be updated and used. Example 4-1 shows a sample LLMD configuration file that is adopted in our system. Here we configure our LLMD receiver to join a multicast group 239.100.200.199 to subscribe a message from the CVT_TO_MSGSTORE topic whose listening port is 34343. To run the LLMD server, see Chapter 5, “Test scenarios” on page 73.

**Example 4-1   LLMD configuration file**

```plaintext
# Licensed Materials - Property of IBM
# 5724-T21 IBM WebSphere MQ Low Latency Messaging
# (C) Copyright IBM Corp. 2009 All Rights Reserved.
# # US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with
# IBM Corp.
# Log File Name
logfile llmd.log
# Log Level
loglevel debug
# Log Filter
logfilter debug

#init the sample readback library. Uncomment the line below to load
#the llmdreadback library during startup. Note: llmdreadback library
#must be in the path.
#init readback init_readback

#Server Control Port
controlport 16100
#Allow the access to client
allow 127.0.0.1
# Enable or Disable Coordinated Cluster Time (CCT)
# Set 1 to enable CCT for Message Store Server only. 2 to enable the CCT for both
# Message Store Server and LLM instance(s).
#cctlevel 0

# The topic mapper needs to be defined before the LLM instance.
```
4.3 Persisting data in Coordination Manager

To feed the messages into the message store server, an RMM transmitter can be provided by application to multicast the messages to the group that the message store server is subscribed to. We need to add the multicast group and the topic from the message store server configuration into the mmtransmitter configuration file.

4.3.1 Configuration file

Example 4-2 shows the configuration of a transmitter to persist data into the msg store, as can be seen in packet_converter.cfg in our sample programs. The parameter value of topic and dataport and MCG should be consistent with the LLMD configuration file.

Example 4-2  Transmitter configuration to persist data into message store

```
[TierTxConfig]
Name=Cvt1_TierTx
ID=0
DataPort=34343 # the data port which msg store receiver listens to.
AdvancedConfigFile=none
NICIP=127.0.0.1
Protocol=1
```
MaxMemoryAllowedKBytes=256000
MinimalHistoryKBytes=128000
PacketSizeBytes=8192

[TierTxTopic]
ID=0
Name=CVT_TO_MSGSTORE # the topic name which msg store server subscribes to
MCG=239.100.200.199 # the multicast group which msg store server subscribes to.
useNak=1
Acker=0

4.3.2 Sample code

Example 4-3 demonstrates a sample code to transmit data into Coordination Manager. To run the samples, see Chapter 5, “Test scenarios” on page 73.

Example 4-3   Code of transmit data to Coordination Manager

/* converter_packet.c */
int sequenceNumber=pHeader->seqnum;
    int expCode = 0;
    char myErrDesc[256];

    // 0. copy into the Tx Message
    memcpy(converter_tx_msg.msg_buf, payload, pHeader->packet_length);

    rc = rmmTxSubmitMessage(&converter_topic_t, &converter_tx_msg, &expCode);

    if ( rc != RMM_SUCCESS)  
    {
        ...
    }  
else {

    /*caculate the timestamp that current msg is saved into msg store 
    and keep the timestamp value into share memory for retransmit use. */

    if(((sequenceNumber%minsamplingseqnum)==0)&&(sequenceNumber<maxmsgnumperday))
    {
        pthread_mutex_lock(&llmtshm_lock);
        write_llmts_shm(sequenceNumber/minsamplingseqnum,
llm_getCurrentTime()+difftimestamp);
        pthread_mutex_unlock(&llmtshm_lock);
    }

}
4.4 Querying persisted data

There are two typical methods of retrieving data from the Coordination Manager Server.

4.4.1 Retrieval data from Coordination Manager

This section discusses the types of retrieval possible.

Retrieval by timestamp
To develop an application to retrieve data from Coordination Manager, we follow these steps with the use of a message store API listed in Table 4-2.
1. Create TCP channel to the LLMD.
2. Use its own asynchronous callbacks to overwrite on messages and on events of LLM.
3. Use llm_getStoredMessages or llm_getStoredMessagesLocal to retrieve the data with specified time range from Coordination Manager.

Table 4-2  Message store API by timestamp

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>llm_defineConnection()</td>
<td>Open an tcp connection to Coordination Manager</td>
</tr>
<tr>
<td>llm_getStoredMessages</td>
<td>Get stored messages from a remote message store. It will get all stored messages and events within the specified range and call the on_msg and on_event callback functions.</td>
</tr>
</tbody>
</table>

Retrieval by sequence of LLM
As before, we follow these steps to retrieve data by LLM sequence from the Coordination Manager with the use of the LLM API functions listed in Table 4-3:
1. Create a TCP channel to the LLMD.
2. Use its own asynchronous callbacks to overwrite on messages and on events of LLM.
3. Use llm_getStoredMessagesPos or llm_getStoredMessagesLocalPos to retrieve the data with a specified sequence range from the Coordination Manager.

Table 4-3  Message store API by sequence

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>llm_getStoredMessagesLocalPos</td>
<td>Get stored messages by running as a Coordination Manager extension. This will get all stored messages and events within the specified range and call the on_msg and on_event callback functions.</td>
</tr>
<tr>
<td>llm_getStoredMessagesPos</td>
<td>Get stored messages from a remote message store from a position. This will get all stored messages and events within the specified range and call the on_msg and on_event callback functions.</td>
</tr>
</tbody>
</table>
4.4.2 Sample code

In this paper, we provide an example to demonstrate how to retrieve the message from msgstore by the message sequence number defined in the application. At some time, we require to retransmit all those messages with a specified sequence number in the application. The packet converter and retransmitter component in our system will transport messages to the Coordination Manager.

The code of both the samples is available [code directory]. The logic of the two components is described next, referring to the code samples within Example 4-4 and Example 4-5 for details. To run the samples, see Chapter 5, “Test scenarios” on page 73.

Packet convert

The packet convert component performs the following steps:

1. Subscribe the topic to receive the published message from publisher. Then create an RCMS transmitter to send data to the remote Coordination Manager to persist trade data.

2. Filter the irrelative messages by message type defined.

3. Calculate the time differential between application clock and Coordination Manager clock, and get the timestamp that the message persisted into Coordination Manager. Create a shared memory table (see the description in Table 4-4) to keep the specific ilm_t timestamp indexed by application message sequence number.

Table 4-4  sequence and timestamp map table

<table>
<thead>
<tr>
<th>Field description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Index</td>
<td>int64_t</td>
<td>sequencenum/munsamplingnum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, if munsamplingnumber =10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The index of 1 represents sequence number 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The index of 10 represents sequence number 100</td>
</tr>
<tr>
<td>Timestamp of message persisted into CM</td>
<td>int64_t</td>
<td>Timestamp of message persisted into CM</td>
</tr>
</tbody>
</table>

Example 4-4  pack_converter.c

/* Get time differential between application clock and Coordination Manager clock*/

void cactimediff_CM()
{
    if (!conn) {
        conn = llm_defineConnection(ipaddr, NULL);
        ...
    }

    t1 = llm_getCurrentTime();
    /*Get Coordination Manager clock*/
    t2 = llm_getConnectionTime(conn);
    t3 = llm_getCurrentTime();
    difftimestamp = t2 - (t1+t3)/2;
    ...
}

/*call back of on_packet function*/
void converter_on_packet(rmmPacket *packet,void *user)


```c
T_OutgoingPacketHeader* pHeader = (T_OutgoingPacketHeader*) payload;
...

pHeader->product_type = EQT_LEVEL_1;
pHeader->msg_count = packet->num_msgs;
pHeader->packet_length = OUTGOING_HEADER_SIZE;     // length of packet
    inclusive of packet header

for(i=0; i < packet->num_msgs; i++)
{
    rmmRxMessage *message = (rmmRxMessage *)(&packet->msgs_info[i]);
    pMEMsg = (T_ME_MSG*)message->msg_buf;
    if ( i == 0)
    {
        pHeader->seqnum = pMEMsg->msgSeqNum;
    }
    if(!i==0)
    {
        memcpy( &(payload[OUTGOING_HEADER_SIZE]), pMEMsg, pMEMsg->Len);
        pHeader->packet_length = OUTGOING_HEADER_SIZE+pMEMsg->Len;
    }
    else
    {
        memcpy( &(payload[pHeader->packet_length]), pMEMsg,
        pMEMsg->Len);
        pHeader->packet_length += pMEMsg->Len;
    }
}
}
```

---

**Retransmit**

The retransmitter component performs the following steps:

1. Accept the input of the application start sequence and end sequence from the standard input (stdin).
2. Query the shared memory created by packeter converter and get the corresponding llm_t timestamp by the input sequences.
3. Open a tcp connection to Coordination Manager and invoke get LLMI API to retrieve the data.
4. When retrieving a message from Coordination Manager, the retransmitter will filter the message out of input sequence range and send out the right message to external UDP multicast subscribers such as feed handles.
Example 4-5  retransmit.c
/* map the input sequence to llm time stamp by searching the shared memory */

int mapseq2llmt(int64_t startseqnum, int64_t endseqnum, int64_t *starttime, int64_t *endtime)
{
    int i=0;
    *starttime = read_llmts_shm(startseqnum/minsamplingseqnum);
    *endtime = read_llmts_shm(endseqnum/minsamplingseqnum);
    return 0;
}

int processMSbytime(llm_connection_t * conn, inputdata_t * input, int64_t *fromtime,
int64_t *totime, userdata_t * output)
{
    ...
    rc = create_conn_getSM(conn, input->serverip, input->policy,input->domain, output,
*start_p, *stop_p);
    ...
    return rc;
}

**
* Create connection and get the stored message from the message store.
*
*/
int create_conn_getSM(llm_connection_t * conn, char * ipaddr, char * policy, char * domain, userdata_t * output, llm_time_t start, llm_time_t end) {
    int    rc = 0;
    const char * errstr, * repl;
    if (!conn) {
        conn = llm_defineConnection(ipaddr, NULL);
        ...
    }
    /*Calling the Message Store API to get the stored messages*/
    rc = llm_getStoredMessages(conn, domain, policy, NULL, on_msg, cb_on_event, (void *)
output, start, end);
    ...
    /*Free Connection*/
    if(conn!=NULL){
        1lm_freeConnection(conn,NULL);
    }
    return 0;
}

int on_msg(llm_msg_t * msg, void * userdata) {
    ...
    if(msg) retransmit(msg);
void retransmit(llm_msg_t * msg)
{
    T_OutgoingPacketHeader* pHeader = (T_OutgoingPacketHeader*) msg->message;
    if(dofiltermsg(pHeader->seqnum))
    {
        printf("skipped sequence number %d timestamp %llu\n",pHeader->seqnum,msg->timestamp);
        return ;
    }
    size_t byteCount = sendto(converter_sock, pHeader, pHeader->packet_length, 0, (struct sockaddr *)&converter_mcastGroup, sizeof(converter_mcastGroup));
}

main()
{
    if(open_seqllmts_shm(llmtshmkey)==0)
    {
        mapseq2llmt( startseqnum,endseqnum,&fromtime,&totime);
        /*Send request to the message store server*/
        processMSbytime(conn, input,&fromtime, &totime, msoutput);
    }
}

4.5 Summary

In this chapter, we have talked about the infrastructure of market data distribution in stock exchange industry. We introduce a few objects inside the infrastructure which we can see as bellow. It is easy for us to implement your own infrastructure of market data distribution based on our sample codes.
4.5.1 Market data publisher

This component provides the following functions:

- Filters on message type base on market data product definition.
- Leverages LLM packetization to create natural payload.
- Provides market data feed sequence number.

4.5.2 UDP format converter

This component provides the following functions:

- Converts LLM packet/messages into a single UDP packet for internet distribution.
- The same UDP packet becomes one LLM message to be stored in Message Store.

4.5.3 Retransmission

This component provides the following functions:

- Retrieves stored message (previously sent UDP packet) to support market data retransmission.
- Source for delayed market data feed.
Test scenarios

In this chapter, we discuss our test scenarios.

There are many key considerations for enterprise integration, but two of them—performance and availability—become critical when you think of using a solution such as WMQ-LLM.

Because a performance load test is a significant investment, you will want to carefully analyze the requirements that identified the need for testing. Unfortunately, the runtime performance and bottlenecks of a system are not easily simulated in the lab. On the other hand, carefully planned test scenarios must be able to validate whether the proposed solution will scale and perform appropriately, given hardware plans.

Many high availability implementations focus on failover and disaster recovery. However, a system must also ensure continuous capacity and performance as defined by its business requirements.
5.1 Building the test scenarios

The performance project will be staged as a series of performance scenarios. Each test scenario is designed to answer specific questions. The result of each test must confirm or disprove your hypotheses, narrowing the choice of optimal design alternatives, which then helps to define the next test.

You can outline the global objectives and questions to be answered, then sketch a brief plan for each test indicating what will be manipulated and what will be measured. Finally, you analyze and record the results, update the picture of what you know versus what you need to know, and zero in on objectives for the next round of tests.

5.1.1 Independent variables

The first test design question you need to address is what factors you can manipulate for comparison. These include anything that you have control over, that represent a realistic alternative in deployment, and that are likely to have bearing on the performance, scalability, or reliability of the system. Some common candidates include these:

- The distribution of components across machines
- The mapping of component connections, sessions, and threads
- Provider of service hosting technology
- Clustering and failover options
- Alternative implementations of key services, such as routing and transformation

5.1.2 Dependent variables or measures

For each test case, the test software needs to measure key factors in the outcome, such as these:

- Throughput: The number of messages, service invocations, or round-trip process executions that can be performed in a period of time.
- Latency: The time required for a message delivery or complete service round trip.
- Exceptions: The count of failures and rejected messages that occur during the test.

These measures must apply directly to the cost/benefit objectives defined in the benchmark analysis. They enable you to scale the actual value of measured improvements in throughput, latency, simplicity or quality and, therefore, prioritize the design alternatives.

In addition, you will want to measure co-variants of the results, specifically the finite resources being consumed from the underlying infrastructure. These measures are especially important if you want to do long-term resource planning for the integration infrastructure. Important co-variant measures include these:

- CPU usage and multi-CPU scalability
- Memory working set
- Thread contention and thread context switches
- Network load
- Disk I/O rates
5.1.3 Control variables or constants

There are other variables that are not relevant to the benchmark objectives and that are best kept constant. These include factors that you might have no control over, perhaps hardware network speed or the rate of an incoming feed. They might include choices that are already made and therefore are not open to change, such as operating system, application server setup, or message schema and size. Finally, they might also include potential artifacts in the test that are not likely to occur in the production environment and are, therefore, undesirable in the performance simulation.

5.2 Configuration files and script locations

This section explains the paths where the resources for the project are located. We will use the variable \textit{install\_path} to refer to the installation path for the product.

The project contains the following resource folders:

\begin{itemize}
  \item bin: Contains the execution program of all the components
  \item doc: Contains documentation of the project
  \item src: Contains source code of the project as well as file configuration for compiling the code
  \item test: Contains the configuration files of all the components
\end{itemize}

Every component comes with a default script to run the sample. You can change parameters in configuration files to customize behavior of the components. All the scripts to run the samples are in install\_path/test.

Table 5-1 shows the scripts and configuration file corresponding to each component.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Component & Scripts & Configuration file \\
\hline
Order Client & client01.sh & gw_publisher01.cfg \\
 & client02.sh & gw_publisher01.cfg \\
Order Gateway & run_gateway01.sh & gw_publisher01.cfg \\
 & run_gateway02.sh & gw_publisher01.cfg \\
Matching Engine & me1.sh & me1.cfg \\
 & me2.sh & me2.cfg \\
Market Data Publisher & md_publisher.sh & md_publisher.cfg \\
Format Converter & packet_converter.sh & packet_converter.cfg \\
Retransmitter Service & retransmit.sh & retransmit.cfg \\
Message Store & run_msgstore.sh & llmd.cfg \\
Subscriber Client & socket_tester.sh & \\
\hline
\end{tabular}
\caption{Scripts and configuration files of sample components}
\end{table}
5.3 Running the sample components

In the following sections, we introduce the way you can run each test component. It is important to understand that, when you are going to perform some kind of test, you need to follow a specific procedure in order to have a similar behavior across the tests and over several tests.

5.3.1 Running the Order Clients

The startup of this component includes the following steps:

1. Make sure the same component is not running on the machine:
   
   $ ps -ef | grep client

2. Start the Order Client components:
   
   $ ./client01.sh
   $ ./client02.sh

3. After both order clients are running, you will be able to start the sending of messages by typing s key. The output of client01 is illustrated in Example 5-1; note these facts:
   
   a. We are simulating different Order Clients. This means that, although Client 01 and Client02 send the same type of messages, both are expecting a reply from each message that each sent.
   
   b. Each order client is connected to a different topic.
   
   c. Both clients use RMM over TCP.

   **Example 5-1**  order client01 sending 100 messages

   [redpaper@saw123-sys1:CLIENT test]$ ./client01.sh
   *** Event Received: [MBU_ACTIVE] 72 T:CLI01_TO_GW01 S:127.0.0.1 P:50444 Parm1:
   Destination was added to the list for stream.
   base time of day since 1970 (seconds):    1302813295.781882
   base time since calling initTime (seconds):             0.781949
   ****Batch: 100 messages****

   Client Ready. [s] to start sending test orders. [n] to quit.

4. After the test finishes, depending on the latency method used, you have different latency metrics. Note these considerations:
   
   a. Read what was explained in “Clock synchronization” on page 14.
   
   b. If all phases of the latency option are used, you will see the latency of all the samples on every hop. You are able to configure the start and end latency phase. The output of client01 is illustrated in Example 5-2.

   **Example 5-2**  Latency of the messages over all phases between Clients and MEs

   ==============================================================
   ***Latency of all phases***
   Phases                          SampleCount         Min
   Max  Avg  30+ms   Wrong   30+ms
   TSCLIENTGWOUT -> TSCLIENTGWIN   1000              55
   368  93   0        0
   ==============================================================


### Chapter 5. Test scenarios

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Count</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS_CLIENT_GW_IN</td>
<td>TS_GW_MEC_OUT</td>
<td>1000</td>
<td>11</td>
</tr>
<tr>
<td>TS_GW_MEC_OUT</td>
<td>TS_GW_MEC_IN</td>
<td>945</td>
<td>264</td>
</tr>
<tr>
<td>TS_GW_MEC_IN</td>
<td>TS_MEC_MX_IN</td>
<td>1000</td>
<td>12</td>
</tr>
<tr>
<td>TS_MEC_MX_IN</td>
<td>TS_ME_GW_OUT</td>
<td>1000</td>
<td>71</td>
</tr>
<tr>
<td>TS_ME_GW_OUT</td>
<td>TS_ME_GW_IN</td>
<td>1000</td>
<td>761</td>
</tr>
<tr>
<td>TS_ME_GW_IN</td>
<td>TS_GW_CLIENT_OUT</td>
<td>1000</td>
<td>7</td>
</tr>
<tr>
<td>TS_GW_CLIENT_OUT</td>
<td>TS_GW_CLIENT_IN</td>
<td>1000</td>
<td>65</td>
</tr>
</tbody>
</table>

[i. If you used the round trip latency option, you see the latency of the entire round trip through all the phases where the message passed. The output of client01 is illustrated in Example 5-3.

**Example 5-3  Latency of the entire round trip where message passed**

<table>
<thead>
<tr>
<th>Latency between points: TS_CLIENT_GW_OUT - TS_GW_CLIENT_IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleCount: 81</td>
</tr>
<tr>
<td>Min(us) 591</td>
</tr>
<tr>
<td>Max(us) 44813164</td>
</tr>
<tr>
<td>Avg(us) 2324</td>
</tr>
<tr>
<td>Median(us) 885</td>
</tr>
<tr>
<td>StdDev(us) 5227</td>
</tr>
<tr>
<td>95th(us) 7364</td>
</tr>
<tr>
<td>99th(us) 24643</td>
</tr>
<tr>
<td>99.9th(us) 24643</td>
</tr>
<tr>
<td>5ms 7</td>
</tr>
<tr>
<td>10ms 5</td>
</tr>
<tr>
<td>30+ms 919</td>
</tr>
<tr>
<td>Wrong 0</td>
</tr>
</tbody>
</table>

### RX Metrics:
- MmsgRecvd:1000 BytesRecvd:0 PktsRecvd:348940 PktProc:0
- PktsLost:1010 PktsDrop:0 PktsDupl:998
- RepairPktsReq:0 RepairPktsRecvd:0
- MaxRatePkts:0 MaxRateKbps:0

### TX Metrics:
- MmsgSent:1000 BytesSent:0 PktsSent:348000
- Buffers:0 BuffersInUe:1000
- RepairPkts:0 NAKs:25574 ACKs 1002
- MaxRatePkts:0 MaxRateKbps:0
To stop the components, follow these steps:
1. Make sure the same component is not running on the machine.
2. Observe the latency introduced by the failure.
3. Restart the terminated ME processes.
4. Wait for RCMS to synchronize the new processes.
5. Observe the latency introduced by the synchronization process.
6. Go back to step 1.

5.3.2 Running the Order Gateways

To start up this component, follow these steps:
1. Make sure the same component is not running on the machine:
   $ ps -ef | grep client
2. Start the Order Client components:
   $ ./client01.sh
   $ ./client02.sh
3. After both order clients are running, you will be able to start the sending of messages by typing the “s” key. The output of client01 is illustrated in Example 5-4.

Note these considerations:
   a. We are simulating different Order Clients, meaning that, although client01 and client02 send the same type of messages, both are expecting a reply from each message that each one sent.
   b. Each order client is connected to a different topic.
   c. Both clients use RMM over TCP.

Example 5-4  order client01 sending 100 messages

[redpaper@saw123-sys1:CLIENT test]$ ./client01.sh
*** Event Received: [MBU_ACTIVE] 72 T:CLI01_TO_GW01 S:127.0.0.1 P:50444
Parm1: Destination was added to the list for stream.
base time of day since 1970 (seconds):    1302813295.781882
base time since calling initTime (seconds):             0.781949
****Batch: 100 messages****
Client Ready. [s] to start sending test orders. [n] to quit.

d. After the test finishes, depending on the latency method used, you have different latency metrics. Note these considerations:

e. Read the explanation in “Clock synchronization” on page 14.

f. If all phases of the latency option are used, you will see the latency of all the samples on every hop. You are able to configure the start and end latency phase. The output of client01 is illustrated in Example 5-5.

Example 5-5  Latency of the messages over all phases between Clients and MEs

============================================================================
***Latency of all phases***
<table>
<thead>
<tr>
<th>Phases</th>
<th>SampleCount</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>Avg</td>
<td>30+ms</td>
</tr>
</tbody>
</table>
-------------------------------------------------------------------------------
ii. If you used the round trip latency option, you will see the latency of the entire round trip through all the phases where the message passed. The output of client01 is illustrated in Example 5-6.

**Example 5-6  Latency of the entire round trip where the message passed**

---

Latency between points: TS_CLIENT_GW_OUT - TS_CLIENT_GW_IN

| SampleCount: | 81 |
| Min(us)     | 591 |
| Max(us)     | 44813164 |
| Avg(us)     | 2324 |
| Median(us)  | 885 |
| StdDev(us)  | 5227 |
| 95th(us)    | 7364 |
| 99th(us)    | 24643 |
| 99.9th(us)  | 24643 |
| 5ms         | 7 |
| 10ms        | 5 |
| 30+ms       | 919 |
| Wrong       | 0 |

---

**RX Metrics:**

- MsgsRcvd:1000
- BytesRcvd:0
- PktsRcvd:348940
- PktProc:0
- PktsLost:1010
- PktsDrop:0
- PktsDupl:998
- RepairPktsReq:0
- RepairPktsRecvd:0
- MaxRatePkts:0
- MaxRateKbps:0

---

**TX Metrics:**

- MsgsSent:1000
- BytesSent:0
- PktsSent:348000
- Buffers:0
- BuffersInUe:1000
- RepairPkts:0
- NAKs:25574
- ACKs:1002
- MaxRatePkts:0
- MaxRateKbps:0

---
To stop the components, these are the steps:
1. Make sure the same component is not running on the machine.
2. Observe the latency introduced by the failure.
3. Restart the terminated ME processes.
4. Wait for RCMS to synchronize the new processes.
5. Observe the latency introduced by the synchronization process.
6. Go back to step 1.

5.3.3 Running the Market Data Publisher

Startup the publisher component by running the command line in Example 5-7 and press the Enter key. You get the following output as shown in Example 5-8.

Example 5-7    Command line to launch publisher

[redpaper@saw123-sys4: PUB test]$ cat md_publisher.sh
#!/bin/bash
../bin/md_publisher -c ../test/md_publisher.cfg -verbose
bash-3.2$/md_publisher.sh

Example 5-8    Output of running publisher

message sequence number: [831] len [264]
sent message sequence number: [832] on topic [PUB_TO_WORLD]
message sequence number: [832] len [264]
sent message sequence number: [833] on topic [PUB_TO_WORLD]
message sequence number: [833] len [264]
sent message sequence number: [834] on topic [PUB_TO_WORLD]

5.3.4 Running the Message Store

Follow these detailed steps to start up the LLM-CM Server:
1. Add the LLM_installation_directory/libOS_ARC directory into the LD_LIBRARY_PATH (Linux).

Tip: OS_ARC is either 32 or 64.

2. Go to the LLM_installation_directory/libOS_ARC directory:

Depending on your operating system, type the following commands and press Enter:
./llmd ./llmd.cfg

Tip: Use -d if you want to run LLMD as a background process.

If everything is successful, you will see the output result as in Example 5-9.

Example 5-9    LLMD startup output

Low Latency Messaging Coordination Manager - Version 2.6.0.0 20110315.1
5724-T21 IBM WebSphere MQ Low Latency Messaging
Copyright (C) 2009, 2010 IBM Corp. All rights reserved.
5.3.5 Running the Format Converter

Start up the publisher component by running the command line in Example 5-10 and press the Enter key. We get the following output as shown in Example 5-11.

Example 5-10  Command line to launch converter

```
[redpaper@PUB test]$ cat packet_converter.sh
../bin/packet_converter -c ../test/packet_converter.cfg -verbose
[redpaper@PUB test]$ ./packet_converter.sh
```

Example 5-11  Output of running converter

```
converter: packet seqnum[988], msg_count[1], packet size [292].
message sequence number: [989] len [264]
message sequence index [99]-->LLM time stamp [1302798771057356500]
sent message sequence number: [990] on topic [CVT_TO_MSGSTORE]

converter: packet seqnum[989], msg_count[1], packet size [292].
message sequence number: [990] len [264]
sent message sequence number: [991] on topic [CVT_TO_MSGSTORE]
```

When data is transmitted from the Packet converter to the Coordination Manager, it will create the data file under the msg store directory which we defined in the LLMD configuration. We can check the data file created by the Coordination Manager as Example 5-12.

Example 5-12  Check the data file of Coordination Manager

```
[redpaper@PUB test]$ pwd
/home/redpaper/test/msgstore
[redpaper@PUB test]$ ls -lRh
.
: total 16K
drwxr-xr-x 3 redpaper redpaper 4.0K Apr  6 12:09 CVTTOMSGSTOR_Q7L0M9
-rw-r--r-- 1 redpaper redpaper 28 Apr  6 18:53 store.lock
./CVTTOMSGSTOR_Q7L0M9:
: total 80K
-rw-r----- 1 redpaper redpaper 64K Apr  6 18:56 20110406_160956905

drwxr-xr-x 2 redpaper redpaper 4.0K Apr  6 18:53 ix
./CVTTOMSGSTOR_Q7L0M9/ix:
: total 0
```
5.3.6 Running the Retransmitter Service

Run the command line in Example 5-13 and in our environment we will get the output as shown in Example 5-14.

Example 5-13 Command line to launch retransmitter

```
[redpaper@PUB test]$ ./retransmit.sh
[redpaper@PUB test]$ cat retransmit.sh
#!/bin/bash
#export LD_LIBRARY_PATH=~/LLM/llm25_1229/lib64:$LD_LIBRARY_PATH
rm -rf log/*
../bin/retransmit -startseq 10 -endseq 30 -file ./retransmit.cfg
run_gateway.sh test_publisher.cfg
```

Example 5-14 Output of running retransmitter

```
MCast Address is 224.0.0.37.
the start sequence is [10] map to timestamp [1302536573847437000]
the end sequence is [30] map to timestamp [1302536578515608000]
a invalid message with sequence number 9 timestamp 1302536573864253000 out of sequence range and skipped
retransmit timestamp = 1302536574033537000 packet seqnum[10], packet size [224] byte [224].
...
retransmit timestamp = 1302536577624597000 packet seqnum[26], packet size [224] byte [224].
retransmit timestamp = 1302536577874777000 packet seqnum[27], packet size [224] byte [224].
retransmit timestamp = 1302536578213619000 packet seqnum[28], packet size [224] byte [224].
```

5.3.7 Running the Client Subscriber

Run the command line in Example 5-15 and in our environment we will get the output as shown in Example 5-16.

Example 5-15 Command line to launch subscriber

```
[redpaper@PUB test]$ cat socket_tester.sh
#!/bin/bash
export LD_LIBRARY_PATH=~/LLM/llm25_1229/lib64:$LD_LIBRARY_PATH
rm -rf log/*
../bin/socket_tester -m s -a 224.0.0.37 -p 12345 -i 127.0.0.1 -b 2000 -n 0 -1 1
```
5.4 Integration test cases

In this section, we describe a few scenarios so you can run the sample application that is included in this paper. You can try, on your own, most of the characteristics for RCMS that we have discussed in the previous chapter.

First, we start with a simple scenario, running all the components in a single computer and then we continue with more complex scenarios where the components run in different computers.

### 5.4.1 Scenario 1: A single computer

This scenario is the most basic, all the components are running in the same computer using only RCMS singletons.

The objective is provide a simple guide to show how all the components work together so you can become familiar with all of them before working with more complex scenarios.

**Setup**

Because all the components are running on the same server, we only need to set all NIC parameter to the loopback interface (127.0.0.1). Make sure that multicast traffic is allowed on each interface.

In the directory scenarios/scenario1/test you will find all the configuration files already modified to use the loopback interface. Notice that no changes in topics names nor multicast groups are necessary.

**Running the scenario**

**Script files:** The sample script files are written for 32 bits platform. For 64 bits, just change the following line on each .sh file.

```bash
export LD_LIBRARY_PATH=/opt/ibm/llm/lib64:$LD_LIBRARY_PATH
```
Follow these steps:

1. Make sure that none of the components are already running in the server if it is necessary to reboot the computer.

2. Build the code source:
   a. Change to the directory common/src.
   b. Remove any existing compiled programs.
      
      ```
      $ make clean
      ```
   c. Set up the LD_LIBRARY_PATH according with your platform (32 to 64 bits).
      
      ```
      $ export LD_LIBRARY_PATH=/opt/ibm/llm/libXX
      ```
   d. Compile the code.
      
      ```
      $ make
      ```

3. Use the script shell to run all the components one by one:
   a. Use a different console window to run each component. Start with the client, follow with the order gateway, matching engine, publisher, etc.
      
      ```
      $ ./run_client.sh
      $ ./run_gateway.sh
      $ ./run_me_singleton.sh
      $ ./run_publisher.sh
      $ ./run_msgstore.sh
      $ ./run_packet_converter.sh
      $ ./run_transmit.sh
      ```

4. After every component is running, go to the client’s terminal and in the console press **s** to start the test.

**Validating the test**

For this first scenario, the client is configured to send out ten messages at a rate of one message per second. This option is configurable and you can change it if you want it to, you just need to modify the shell scrip file.

The test must not take too much time to finish. At the end, you see a similar output as in Example 5-17.

*Example 5-17  Output for the test in the client terminal*

$ ./run_client.sh

```
*** Event Received: [MBU_ACTIVE] 72 T:CLIENT_TO_GATEWAY S:127.0.0.1 P:50377 Parm1:
Destination was added to the list for stream.
...
****Batch: 10 messages****

Client Ready. [s] to start sending test orders. [n] to quit.
*** Event Received: [NEW_SOURCE] 10 T:GATEWAY_TO_CLIENT S:127.0.0.1 P:45904 Parm1:
New stream recognized.

s
Start Sending 10 Messages at 1 message per second...
Message [1] sent. Length: [424]. Response: [0]
...
Message [10] sent. Length: [424]. Response: [0]
Message received from GW [10]. Size [424:424]
Batch messages sent to GW.
```
Each message is received by the gateway, Example 5-18 shows the output.

**Example 5-18  Output from the gateway**

```
$.run_gateway.sh
seconds at midnight is [1303448400]
*** config file: [gw_publisher.cfg]
...
Creating RCMS tier...
*** Event Received: [NEW_SOURCE] 10 T:CLIENT_TO_GATEWAY S:127.0.0.1 P:50377 Parm1: New stream recognized.
Creating tier topic [ME_TO_GW]
Listening topic [ME_TO_GW] on [239.100.200.150].
Creating tier topic [GW_TO_ME]
Order Gateway Ready.
Gateway receives a message from Client. size [424:424]
Message [1] sent to ME. Length: [440]. Response: [0]
Gateway receives a message from ME. size [440:440]
Message [1] replied to Client. Length: [424]. Response: [0]
...
Gateway receives a message from Client. size [424:424]
Message [10] sent to ME. Length: [440]. Response: [0]
Gateway receives a message from ME. size [440:440]
Message [10] replied to Client. Length: [424]. Response: [0]
```

Notice that each message is forward to the matching engine. As shown in Example 5-18 and Example 5-19, the matching engine replies the gateway for each message and send a new message to the publisher.

**Example 5-19  Output from the matching engine**

```
./run_me_singleton.sh
...
Creating RCMS tier...
Creating tier topic [ME_TO_PUB]
Creating tier topic [ME_TO_GW]
Creating tier topic [GW_TO_ME]
Listening topic [GW_TO_ME] on [239.100.200.120].
...
*** Message sent to gateway : <1>
*** Messages sent to publisher : <1>
...
*** Message sent to gateway : <10>
*** Messages sent to publisher : <10>
```

In this scenario, no match was found, therefore no trade report was sent to the publisher. On the publisher side, you see an example similar to the output shown in Example 5-20.

**Example 5-20  Output from the publisher**

```
$.run_publisher.sh
seconds at midnight is [1303448400]
*** config file: [../test/md_publisher.cfg]
...
Creating RCMS tier...
Creating tier topic [ME_TO_PUB]
Listening topic [ME_TO_PUB] on [239.100.200.130].
```
Creating tier topic [PUB_TO_WORLD]
message sequence number: [1] len [440]
sent message sequence number: [2] on topic [PUB_TO_WORLD]
...
message sequence number: [10] len [440]
sent message sequence number: [11] on topic [PUB_TO_WORLD]

These messages sent by the publisher are received by the packet_converter which is responsible to store the data using the message store (Example 5-21).

Example 5-21 Output from packet converter

./run_packet_converter.sh
...
Creating tier topic [PUB_TO_WORLD]
Listening topic [PUB_TO_WORLD] on [239.100.200.190].
Creating tier topic [CVT_TO_MSGSTORE]
MCast Address is 224.0.0.37.
message sequence number: [1] len [440]
sent message sequence number: [2] on topic [CVT_TO_MSGSTORE]
converter: packet seqnum[1], msg_count[1], packet size [472].
...
message sequence number: [10] len [440]
sent message sequence number: [11] on topic [CVT_TO_MSGSTORE]
converter: packet seqnum[10], msg_count[1], packet size [472].

At the end, you can run the retransmitter service to retrieve those messages from the message store.

Troubleshooting
If you ran into any trouble and cannot get the components to work together, try rebuilding the code and check that your multicast traffic in the loopback interface is allowed.

Analyzing results
Although at the end of this test you can see statistics for round trip between different components, the main reason for this test is just to learn how all the components work together. We will see more complex scenarios later in this chapter.

5.4.2 Scenario 2: Using four servers, two gateways, and HA

In this scenario, we are going to use four different computers to host the components. The purpose of using more than one computer is get some statistics information about latency.

The technical details of all four computers used on our test are discussed in Chapter 2, “The demonstration environment” on page 9.

Besides using a two-member RCMS tier for the matching engine component, we are going to run in parallel two gateways; each one receives orders from a different client.
Figure 5-1 shows the architecture of this scenario.

**Setup**

Table 5-2 summarizes how the components were set up on our four servers (which is also shown in Figure 5-1). Some components are not shown in Figure 5-1 for simplicity.

**Table 5-2  Distribution of the components**

<table>
<thead>
<tr>
<th>Server</th>
<th>IP address</th>
<th>Components running</th>
</tr>
</thead>
</table>
| Server 1 | 10.0.0.1   | Client 1  
Client 2  
Gateway 1  
Gateway 2 |
| Server 2 | 10.0.0.2   | Matching Engine 1                                      |
| Server 3 | 10.0.0.3   | Matching Engine 2                                      |
| Server 4 | 10.0.0.4   | Market Data Publisher  
Message Store daemon  
Format Converter  
Retransmitter Service  
Client Subscriber |

Keep in mind that the IP addresses are set in the configuration files for each component. Feel free to modify them as needed (we suggest that you change the IP address after you have run this scenario exactly as is presented to make sure everything works as expected.

Do not forget to disable any firewalls between all the computers and check that every NIC is able to support multicast traffic.
Measuring latency between components

There are two options to measure latency between two components:

- The first option is to record the latency directly in the message. That is, every time the message arrives at or leaves a component, a timestamp is recorded and stored inside the message. This means that you need to add attributes for this purpose.

- The second option is leave the message intact and now make each component responsible to record and keep (in memory or any other persistence means) the timestamp for each message that arrives at or leaves the component. This is supposed to have a more complex structure because you need a way to relate each timestamp with each message arriving or leaving.

In our sample application, we decided to use the first option to keep simple the example. You might be thinking that the second option it is probably a better choice for real production scenarios. And you are right; that option makes it easier to display graphs and numbers so you can easily identify a bottleneck caused by a particular component.

Now, because we have chosen the first option to keep track of the latency records, we have modified our message to hold that information. Inside the message (GW and ME types), we use a simple timeval array to keep each timestamp record as shown in Example 5-22.

Example 5-22  Timestamp attribute for each message

```c
struct timeval TS_INFO[NUMBER_TS_PHASES];
```

We identified 19 phases in the flow; each phase corresponds to a trip from the message coming or going out of a component. Table 5-3 details each phase.

Table 5-3  Latency phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Message origin</th>
<th>Message destination</th>
<th>Component where it is measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS_CLIENT_GW_OUT</td>
<td>Client</td>
<td>Gateway</td>
<td>Client</td>
</tr>
<tr>
<td>TS_CLIENT_GW_IN</td>
<td>Client</td>
<td>Gateway</td>
<td>Gateway</td>
</tr>
<tr>
<td>TS_GW_MEC_OUT</td>
<td>Gateway</td>
<td>Matching Engine</td>
<td>Gateway</td>
</tr>
<tr>
<td>TS_GW_MEC_IN</td>
<td>Gateway</td>
<td>Matching Engine</td>
<td>Matching Engine</td>
</tr>
<tr>
<td>TS_MEC_ME_IN</td>
<td>Matching Engine</td>
<td>Matching Engine</td>
<td>Inside matching engine function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TS_ME_GW_OUT</td>
<td>Matching Engine function</td>
<td>Gateway</td>
<td>Inside matching engine function</td>
</tr>
<tr>
<td>TS_MEC_GW_IN</td>
<td>Matching Engine</td>
<td>Gateway</td>
<td>Matching Engine</td>
</tr>
<tr>
<td>TS_GW_CLIENT_OUT</td>
<td>Gateway</td>
<td>Client</td>
<td>Gateway</td>
</tr>
<tr>
<td>TS_GW_CLIENT_IN</td>
<td>Gateway</td>
<td>Client</td>
<td>Client</td>
</tr>
<tr>
<td>TS_ME_PUB_OUT</td>
<td>Matching Engine</td>
<td>Publisher</td>
<td>Matching Engine</td>
</tr>
<tr>
<td>TS_ME_PUB_IN</td>
<td>Matching Engine</td>
<td>Publisher</td>
<td>Publisher</td>
</tr>
<tr>
<td>TS_PUB_CONV_OUT</td>
<td>Publisher</td>
<td>Format Converter</td>
<td>Publisher</td>
</tr>
<tr>
<td>TS_PUB_CONV_IN</td>
<td>Publisher</td>
<td>Format Converter</td>
<td>Data Converter</td>
</tr>
<tr>
<td>TS_CONV_MSGST_IN</td>
<td>Format Converter</td>
<td>Message Store</td>
<td>Format Converter</td>
</tr>
</tbody>
</table>
Every time a message arrives at or leaves the component, the corresponding timestamp is stored in this array. For that purpose, in the latency library we include two functions as shown in Example 5-23.

**Example 5-23  Latency library two functions**

```c
void setupGWTimeStamp(T_GW_MSG* message, int ts_phase);
void setupMETimeStamp(T_ME_MSG* message, int ts_phase);
```

At the end, when the client component is stopped, a report from different phases is printed out by the client program.

**Components for the scenario**

Table 5-4 summarizes how the components were set up on our four servers (which is also shown in Figure 5-1; some components are not shown in the figure for simplicity).

**Table 5-4   distribution of the components**

<table>
<thead>
<tr>
<th>Server</th>
<th>IP address</th>
<th>Components running</th>
</tr>
</thead>
</table>
| Server 1| 10.0.0.1   | Client 1
          |            | Client 2
          |            | Gateway 1
          |            | Gateway 2 |
| Server 2| 10.0.0.2   | Matching Engine 1                      |
| Server 3| 10.0.0.3   | Matching Engine 2                      |
| Server 4| 10.0.0.4   | Market Data Publisher
          |            | Message Store daemon
          |            | Format Converter
          |            | Retransmitter Service
          |            | Client Subscriber

Keep in mind that the IP addresses are set in the configuration files for each component. Feel free to modify them as needed (we suggest that you change the IP address after you have run this scenario exactly as it is presented to make sure everything works as expected.

Do not forget to disable any firewalls between all the computers and check that every NIC is able to support multicast traffic.
Running the scenario

**Multiple computers:** Running the samples in more than one computer makes the test more complex. One of the things you want to be sure of before running the test is that all the computers are clock synchronized. There are several methods to achieve this, but the complete procedure for this task is out of the scope of this paper.

The easiest way to run the test is to replicate the whole sample directory structure on each server. Make sure, as usual, that no firewall is running in the middle and that multicast traffic is allowed over the network.

The config files included in the scenarios/scenario2/test directory are already configured for this scenario. All the proper IP addresses are set up according to Table 5-3. Note that the same topic names and multicast groups are used, this configuration does not change if you run in one or multiple computers.

As stated in Chapter 3, “Developing highly available matching engines with RCMS” on page 37, the matching engine component is the only one that is not set up as an RCMS singleton.

To run the sample, follow these steps:

1. Make sure that none of the components are already running on each server. If necessary, reboot the computer.
2. Transfer the sample code to all computers and build the applications. If for any reason, you need to rebuild them, do not forget to clean the binary objects first.
3. Start each component one at the time. The order is exactly the same as the first scenario, the only different is what you are going to run two clients, two gateways, and two matching engine applications. Remember that each component must be started in its own terminal window or console. Use Table 5-5 as a guide to run the scripts.

**Table 5-5  Commands to run each component**

<table>
<thead>
<tr>
<th>Server</th>
<th>Scripts issued</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.1</td>
<td><code>.run_client01.sh</code>&lt;br&gt;<code>./run_client02.sh</code>&lt;br&gt;<code>./run_gateway01.sh</code>&lt;br&gt;<code>./run_gateway02.sh</code></td>
</tr>
<tr>
<td>10.0.0.2</td>
<td><code>./run_me1.sh</code></td>
</tr>
<tr>
<td>10.0.0.3</td>
<td><code>./run_me2.sh</code></td>
</tr>
<tr>
<td>10.0.0.4</td>
<td><code>./run_publisher.sh</code>&lt;br&gt;<code>./run_msgstore.sh</code>&lt;br&gt;<code>./run_packet_converter.sh</code>&lt;br&gt;<code>./run_transmit.sh</code></td>
</tr>
</tbody>
</table>

Make sure you use the script files from the scenarios/scenario2/test directory. An easy way to find if you are running the right scripts is that run_gatewayXX.sh is not found in the first scenario, neither is run_meXX.sh nor run_clientXX.sh.

4. Start the test from the console of the first client (client 01):
   
   *Press S to start the test*

5. After a few seconds, you start seeing on each console that the messages are received by each component. The messages are similar to the ones seen in scenario 1.
Troubleshooting

First of all, make sure you have run scenario 1. In this case the only difference is that the components are spread around four computers.

As usual, first check that no firewall is active in the network and that multicast traffic is allowed. If the components do not see each other, try to ping all the computers and telnet using the port for the multicast group.

A common problem is that you are not using the same port for the multicast group as for the transmitter and receiver. In this case, it cannot be possible, because the configuration files are already set up with the correct values.

5.4.3 Scenario 3: Testing high availability for the matching engine

Now it is the time to test the high availability and how the application state is synchronized between our two tier members.

The order in which you start the matching engine components is very important for this test. We are going to assume that ME1 is the first one started at the beginning of this test (refer to Table 5-5 on page 90 to locate the corresponding script) and then ME2 is started.

Then the client is starting to sending order messages; the whole idea is to repeat the previous test. At some point, while the client is still submitting orders, you are going to stop ME1.

As soon as ME1 is out of the line, you need to watch on the console for ME2 that a series of messages are printed out saying that ME2 has become the leader of the tier. At that moment, both tier members share the same application state, so no synchronization process needs to occur. During this whole operation, you can expect to notice no impact in the operation of the whole system.

State messages: At the very beginning of the test, exactly when you start ME2, a series of synchronization state messages are printed out on both consoles for tier members. This is normal, but because there are no orders in the order book yet, no state is transferred from ME1 to ME2.

Now it is time to restart ME1. Wait for a few seconds and then restart it; just use the shell script. At this moment you start seeing synchronization messages on both matching engine consoles. As ME1 is rejoining the tier, it needs to acquire the current application state before starting to process orders. So LLM automatically calls the proper synchronization callbacks so the current application state is asked to ME2 (the tier leader) and transferred to ME1. See 3.5.2, “How LLM state synchronization works” to know exactly how this process takes place.

After the synchronization process between both tier members ends, the order processing can start again. At the client side, You can expect to notice a slight delay, but again, not enough to cause an impact in the whole operation.

If you repeat this test, now stopping ME2, You can expect to observe exactly the same behavior.

Memory: As mentioned in 3.5, “State synchronization” on page 51, it is very important that you keep the memory footprint of your application state as small as possible in order to avoid losing messages while the state is synchronized between tier members and the order processing is on hold.
Setup
For this test we are going to use the same setup as in 5.4.2, “Scenario 2: Using four servers, two gateways, and HA” on page 86.

Running the scenario
To start the test, just follow these steps:
1. Restart all the components on each computer. This is done because all the orders sent are already recorded in the order book component. Restarting all the components clears this state of the application.
2. Follow the same steps as the previous test; see Table 5-5 on page 90. Make sure that you first start ME1 on server 1 before starting ME2 on server 2.
3. On the client1 console, press S to start the test.
4. Wait for a few seconds.
5. Press 9 on the ME1 console to shut down this component.
6. Watch on the ME2 console for the change of role messages saying that ME2 has become the tier leader.
7. Restart ME1, this time you see synchronization state messages while the state application is transferred from ME2.
8. Press 9 on the client1 console after all the order messages are sent to stop this component and see the final statistics.

Conclusions
As mentioned earlier, one of the main features of LLM in RCMS is that it provides all the features for high availability.

In this scenario we tested with a two-member tier, but it can be easily extended to become an n-member tier. One of the great advantages of LLM is that after you have a scenario such as this one running, adding more members to the tier is just a matter of configuration. At the end, the whole tier works as a single unit.

5.4.4 Scenario 4: Testing turbo flow labels

So far, we only have run tests using one client at the time. Clearly, in a more realistic scenario, more than one client can be currently sending order messages to a collection of gateways.

As you might remember from 3.6, “Filtering” on page 55, in our sample application, both gateways are using the same RCMS Topic to publish to the ME tier.

Inside each gateway, every message is marked with a turbo flow label so when the ME tier replies back, it uses the same label to mark out the reply message. After the UpdateOrder message is sent back with the turbo flow label, LLM filters it out as each gateway is configured to accept only messages with its own label. No intervention from the programmer is needed to achieve this.

In this test you are going to start all the components as in the second scenario. Then, after client 1 has started to send its order message, you are going to start sending an order message from client 2.

You can expect to observe no duplicated messages on the client1 or client2 side.
Setup
Again, for this test, we are also using the same setup as in 5.4.2, “Scenario 2: Using four servers, two gateways, and HA” on page 86.

Running the scenario
To start the test, just follow these steps. In this case, it does not make any difference which ME component you start first:
1. Restart all the components on each computer.
2. On the client1 console, press S to start the test.
3. Wait for a few seconds.
4. Now press S on the client2 console; this client starts sending order messages.
5. You see that for every message each client sends, a unique reply is received. This means that LLM is filtering the message for each gateway.
6. Press 9 on both client consoles to end the test and display the statistics information.

Conclusions
LLM provides different techniques to filter messages at the transportation level. This relieves programmers from having to write explicit code to achieve the same goal.

By using turbo flow, you can easily filter out those messages that you are not interested in without penalties in performance.

5.5 Summary

In this chapter we have presented scenarios to illustrate characteristics discussed in this paper for LLM. From the most simple scenario to more complex ones, truly the features of LLM can be seen.

At the end, the main purpose of this document is to help the reader to understand these characteristics and show what benefits they can bring in their companies.

Through all these scenarios, we are looking to get the reader familiar with this product, point out some techniques, and provide some deep explanations.

We believe that the WMQ-LLM product can be used in a number of scenarios, and we hope that the reader now has a better understanding of how to apply it.
Detailed information about the test environments
C1: Machines

We used four main IBM eserver xSeries® systems for the distribution and testing of the developed components. Two more eserver systems were used for NIC bonding tests. Table A-1 shows the characteristics of each machine used for the tests.

Table A-1 Machine characteristics

<table>
<thead>
<tr>
<th>Vendor model</th>
<th>IBM eserver xSeries 236</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>1 Dual core Intel® Xeon™ CPU 3.60GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>1M Level 2 cache</td>
</tr>
<tr>
<td>Front Side Bus Speed</td>
<td>200 Mhz</td>
</tr>
<tr>
<td>Memory</td>
<td>4G</td>
</tr>
<tr>
<td>Disk</td>
<td>SCSI storage 74G</td>
</tr>
<tr>
<td>NIC</td>
<td>2 Broadcom Corporation NetXtreme BCM5721 Gigabit Ethernet</td>
</tr>
</tbody>
</table>

C1: Network topology

When building a network, a key consideration is the availability. In the context of networks, availability means that the network must be operable regardless of failures on network devices. Thus, some techniques must be taken to make the network available all the time.
As shown in Figure A-1, a common production networking distribution has separate network elements over several layers.

![Figure A-1 Common production networking distribution](image)

The scope of this paper covered just the access layer, so we ran tests using different network/application configurations. Because we were expecting to have different behaviors on performance of all the components, we had to understand and record the results.

Table A-2 shows the machines and their link types that were used for the tests. Servers mar07{1,3} were used for bonding tests.

<table>
<thead>
<tr>
<th>Hostname</th>
<th>100 MB/s speed</th>
<th>1G/s speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>saw123-sys1</td>
<td>9.42.170.157</td>
<td>10.0.0.1</td>
</tr>
<tr>
<td>saw123-sys2</td>
<td>9.42.171.106</td>
<td>10.0.0.2</td>
</tr>
<tr>
<td>saw123-sys3</td>
<td>9.42.170.164</td>
<td>10.0.0.3</td>
</tr>
<tr>
<td>saw123-sys4</td>
<td>9.42.171.235</td>
<td>10.0.0.4</td>
</tr>
<tr>
<td>mar071</td>
<td>9.3.179.71</td>
<td>10.10.3.11</td>
</tr>
<tr>
<td>mar073</td>
<td>9.3.179.73</td>
<td>10.10.3.13</td>
</tr>
</tbody>
</table>
Tip: Due to time restraints, we did not cover tests using 10G or InfiniBand. Add more information if tests were done with those servers.

Table A-3 shows the network elements that were used on tests.

| Table A-3 Networking elements |
|-------------------------------|-----------------|----------------|
| Element                      | 1000baseT/Full | 1000IB/Full    |

The machines connected through 1 GB of speed were distributed with the following topology (Figure A-2).

Figure A-2 Network topology
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

IBM Redbooks publications

The following IBM Redbooks publication provides additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- **WebSphere MQ Low Latency Messaging Development Guide**, ZG24-1775-00

You can search for, view, or download Redbooks publications, Redpaper publications, Technotes, draft publications, and Additional materials, as well as order hardcopy Redbooks publications, at this website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- **WebSphere MQ Low Latency Messaging Infocenter and persistence information:**
  


  **Access:** A User Name and Password are required to access this website. You must contact your IBM WebSphere LLM representative to gain access.

- **IBM WebSphere MQ Low Latency Messaging V2.6 delivers enhanced management, monitoring, reliability, filtering, and performance optimizations:**
  

- **OpenFabrics Enterprise Distribution (OFED) information and downloads:**
  
  http://www.openfabrics.org/downloads/OFED/

- **Downloading and assembling WebSphere MQ Low Latency Messaging:**
  
Help from IBM

IBM Support and downloads:
ibm.com/support

IBM Global Services:
ibm.com/services
This IBM Redpaper publication demonstrates the application of WebSphere MQ Low Latency Messaging in financial exchanges and brokerages to facilitate ultra high-speed and low latency messaging among critical components such as order entry gateways and matching engines in a highly available fashion.

In this publication, we explain the environment used for our scenario and provide an explanation of the components developed and tested. We demonstrate high availability obtained with WebSphere MQ Low Latency Messaging utilized by some of the most critical components found in a financial exchange.

We explain how to extend our sample application to create a highly available matching engine component using the RCMS API. In addition, we discuss how the state of the application is transferred from the leading tier member when a tier member joins the tier. We demonstrate the RCMS features used to keep messages in the same order, so every tier member receives exactly the same input, which guarantees the same output from the tier. We also explain how to use real synchrony, virtual synchrony, and fast real synchrony, which is only available in LLM 2.6.

We cover the Coordination Manager Server, which is a component of WebSphere MQ Low Latency Messaging that provides a complementary message persistence solution to store messages and events into the data space. The Coordination Manager Server used in conjunction with the client interface libraries (LLMI) provide the ability to store, retrieve, and replay messages from persistent storage.

This publication was designed for an audience of solution architects and chief technical officers (CTOs) in the Financial Sector marketplace, application developers starting out with WebSphere MQ Low Latency Messaging, and the wider messaging community (WMQ, and so on).