Enabling change at the speed of thought

www.fiorano.com

FioranoMQ® 9

Concept Guide

AMERICA'S
Fiorano Software, Inc.
230 S. California Avenue,
Suite 103, Palo Alto,
CA 94306 USA
Tel: +1 650 326 1136
Fax: +1 646 607 5875
Toll-Free: +1 800 663 3621
Email: info@fiorano.com

EMEA
Fiorano Software Ltd.
3000 Hillswood Drive Hillswood
Business Park Chertsey Surrey
KT16 0RS UK
Tel: +44 (0) 1932 895005
Fax: +44 (0) 1932 325413
Email: info_uk@fiorano.com

APAC
Fiorano Software Pte. Ltd.
Level 42, Suntec Tower Three 8
Temasek Boulevard 038988
Singapore
Tel: +65 68292234
Fax: +65 68292235
Email: info_asiapac@fiorano.com

Contents

Chapter 1: Introduction 11

Messaging Fundamentals 11
- JMS Provider 12
- Loosely Coupled Nature of Messaging Systems 12
- Reliable Delivery of Messages 12
- Messaging Domains 12
- Mode of Consumption of Messages 12
- Administered Objects 13
- Sessions 13
- JMS Message 13

Salient Features of FioranoMQ 13
- High Availability 13
- Clustering 14
- XA Support 14
- Scalability 14
- Application Server Integration 14
- Native Runtime Support 14
- Security 14
- Durable Connections 15
- Large Message Support 15
- Hierarchical Topics 15
- HTTP Support 15
- Logging Facilities 15
- Message Snooping 15
- Dead Message Queue 15
- Encryption, Compression Support 16

Samples 16

Chapter 2: Configuration Concepts 17

Fiorano Component Model 17

Allows the invocation of operations and displays changes in exposed configuration parameters of a component 18

Deployment Profile 19

Default Profiles 20

Configuration Tools 21

Chapter 3: Connection Management 22

Socket Acceptors 22
    Port Number 22
    Protocol 22
    Thread Management 22
    Security Parameters 23
    Configuration 23
Connection Factory 23
    Obtaining a Connection Factory Instance 24
    JNDI Lookup 24
    Creating a new instance 24
Lookup 24
JMX 25
    RMI Connector 25
JMS Connector 25
Pinging 25
    When to Enable Pinging 25
    Salient Features 26

Chapter 4: HTTP Support 27

Client Side Changes 27
Using Proxies 27
    Proxy Authentication 27
Tunneling through Firewalls 28
    Tunneling through SOCKS Proxy Server 28
    Enabling JMS Applets to Tunnel through SOCKS Proxy Server 29
    Additional Notes on SOCKS 29
HTTP Pinging 29

Chapter 5: FioranoMQ Security 30

User Identification and Authentication 30
Data Protection 31
Authentication Based on Digital Certificates 31
Security Realms 31
FioranoMQ User Management 32
    Access Control Management 32
Default Realm 33
NT Realm 33
    Salient Features 33
    Limitations 34
    Troubleshooting 34
RDBMS Realm 35

LDAP Realm 35
Configuring the LDAP Security Realm 36
Miscellaneous Features 36
XML Realm 37
Caching Realm 37
FioranoMQ Security - Salient Features & Advantages 37
Design Advantages 37
Effective Protection of JMS Destinations 37
Centralized Control 38
Destination-based Security 38
Authorization and Access Control 38
Default Users, Groups and ACLs 39

Chapter 6: FioranoMQ Data Stores 40

Storage type 40
File-based store versus JDBC-compliant RDBMS store 40
Default Destinations for sample applications 41
Creating a Default Database 42
Clearing a Database 42

Chapter 7: Managing Administrated Objects 43

Naming Services 43
File 43
XML 43
LDAP 44
RDBMS 44
Cache 44
Salient Features 45

Chapter 8: Message Expiry 46

Point of Checking of Message Expiry 46
On Detection of an Expired Message 46
Dead Message Queue 46
DMQ Configuration 47
Selectively disabling DMQ for a message 47
Message Expired Notifications 47
Configuration 47
Additional points 48

Chapter 9: Snooper 49

Snooper Configuration 49
Working of Snooper 50
Security Settings 50
Miscellaneous Features 50

Chapter 10: Durable Connections 51

Overview 51
Working of Durable Connection 52
Producer on a Durable Connection 52
Consumer on a Durable Connection 52
Advantages 53
- Network Reliability 53
- Store and Forward Capabilities 53
- Transparent Reconnection Code 53
- Message Browsing of Persisted Messages 53
- No Vendor Lock-in 53

Enabling Durable Connections Support 53
Client side Message Cache 54
Serverless Environment 55
- Sample Application 55

Relationship with Revalidate 56
Relationship with CSP 56
Constraints in Durable Connections 57

Chapter 11: Hierarchical Topics 58

Need For Hierarchical Name-Spaces 58
Name Space Notation 58
Creating Hierarchical Topics 59
- Case Insensitive 59
- Spaces in Names 59
- Empty String 59
- Unlimited Length of Topic Names 59
- Unlimited Depth of Topic Hierarchy 59
- Wild Card Support 59

Dynamic Creation of Topics in Hierarchy 60
Looking up Hierarchical Topics 60
Publishing on Node(s) in Topic Hierarchy 60
Subscribing to Node(s) in Topic Hierarchy 60
- Template Characters Used in Subscription 61

Deleting a Hierarchical Topic 63
Publish/Subscribe Across Servers 64

Security Considerations on Hierarchical Topics 64
Limitations 64

Chapter 12: Message Encryption 65

Base Implementation 65
Message Encryption Characteristics 66

Chapter 13: Message Compression 67

Base Implementation 67
Message Compression Characteristics 68

Chapter 14: FioranoMQ Clustering 69

Common Problems of Real-World Systems 69
- Client Unable to Connect 69
- Connection if the Server is Lost 69
- The Server Runs Out of Resources 69
- The Server Goes Down Altogether 70

FioranoMQ: The Solution 70
- Automatic Failover Protection 70
- Transparency and Code Portability 70
- Configurability 70
- Admin System 71
- Connection if the Server is lost 71
- Server Runs Out of Resources 71
- Server's Connection if a Client is Lost 72
- Server-to-Server Communication 72
- Scalability 72

Clustering Components 72
- Dispatcher 73
- Preferred Server 73
- Configuration Parameters 74
- Repeater 75
- Salient Features 75
- No Changes in the Client Application 76
- Robustness in Handling Network Failures 76
- Subscription Mode and Choice of Selectors 76
- Request/Reply Across Repeater 77
- Dynamic Replication Links 77
- Repeater with Load Balancing 77
- Repeater Link 77
- Connection Information 77
- Link Topic Information 78

Configuration Parameters 78
Wild Character Support 79
Dynamic Link Propagation 79
Bridge 80
Bridge Architecture [........ 80
Forwarding Messages to Remote Queues 81
Bridge Features 82
Bridge Configuration 83
Property Name Description 83
Link Properties 84
SourceServer 84
TargetServer 84
Connection Info Properties 84

Chapter 15: Large Message Support 86

Salient Features 86
Reliable transfer of large messages 86
No increase in cache/JVM heap size required 86
The Large message transfer is not restricted to any queue or topic 86
Resume function at both sender and receiver end] 86
Minimal changes in the application code 86
Using Fiorano LMS to Transfer Large Files 87
Message Creation 87
Starting the Message Transfer 87
Tracking the Message Transfer 87
Handling Exceptions During Message Transfer 87
Resuming the Message Transfer 88
Salient Features 88
Consumer Discovery 88
Fragment Size 88
Window Size 88
Sequencing 89
Handling Duplication 89
Handling lost fragments 89
Optimizing Large Message Transfer 89
Fragment Size 89
Window Size 90
Status message frequency 90

Chapter 16: High Availability 91

FioranoMQ's HA - An overview 91
HA Components 92
Backup Server 92

Server States 92
Intra-Enterprise Server Communication 93
Common Persistent Message Store 93
Common Admin and Security 93
Gateway Machine 93
FioranoMQ HA Salient Features 94
Shared and Replication of Databases 94
Application Failover 94
Data Store Consistency (maintained between server switches) 94
Expensive HA Hardware Not Required 94
Implementing a Cluster 95
HA Example Scenario 95
State - 1 (Normal Operation State) 95
State - 2 (Active Server goes down) 96
State - 3 (Backup Server resumes operations) 97
Limitations of HA 98

Chapter 17: Distributed Transactions 99

Introduction 99
Use Case 100
Transactions and the Distributed Transaction Processing (DTP) System 101
Components of a Distributed Transaction 101
FioranoMQ as a Distributed Transaction Resource Manager 102
Transactions with J2EE 103
FioranoMQ XA Implementation Notes 104
Limitations of XA Implementation of FioranoMQ 105

Chapter 18: FioranoMQ Content Based Routing 109

FioranoMQ Content Based Routing 109
Using FioranoMQ Content Based Routing 111
Setting up the FioranoMQ Server for CBR 111
FioranoMQ CBR XPath Support 111
Publishing XML Messages 112
Subscribing to XML Messages 114
CreateDurableSubscriber 114
CreateSubscriber 114
Handling Massive Number of Subscribers 117
XML Support 118
FioranoMQ Content - Based Message Selector Language 120
General Form 120
Subset of Supported XPath Queries 121
Identifiers 122
Operators 122

Literals 123
Example XMLs 124
Elements Only XML 124
Attributes Only 125
Elements and Attributes XML 126
Limitations of FioranoMQ Content Based Routing 127

Chapter 1: Introduction

The exchange of events based messaging data is an important function of business enterprises. Messaging comprises of communication between software applications or between objects in distributed system.

Fiorano's messaging solution is based on JMS 1.1 standards for enterprise messaging.

FioranoMQ® (FMQ) is a high-performance, stable, and secure Java implementation of JMS. Fiorano’s messaging solution reduces the development time of applications requiring a messaging infrastructure. The automatic store-and-forward capability of Fiorano’s messaging solution across multiple servers ensures high scalability, high availability as well as high performance of message delivery which is across the faulty networks.

Distributed transaction support within FioranoMQ ensures high levels of consistency and reliability of message delivery; this aids the process of developing and deploying internet, intranet and extranet applications.

FioranoMQ is JMX-enabled making it easy for administrators to manage and monitor its server. FioranoMQ includes libraries written in C, C++ and C# for all major platforms. These native runtime libraries allow non-java clients to talk directly to the java server and exchange information with JMS-compliant clients.

FioranoMQ security includes integrated JSSE support. The Java Secure Socket Extension (JSSE) enables secure internet communication using Java SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols. Developers can, therefore, enable secure data transfer between client and server.

Messaging Fundamentals

This section gives an overview of terminologies and processes within messaging systems: JMS Provider

- Loosely coupled nature of Messaging Systems
- Reliable Delivery of Messages
- Messaging Domains
- Mode of Consumption of Messages
- Administered Objects
- Sessions
- JMS Message

JMS Provider

In messaging systems, clients connects to message providers. Each message has a specific destination which is maintained by the provider. Functions of client applications include producing messages for specific destinations and receiving messages from specific destinations. The provider is responsible for the routing of messages.

Loosely Coupled Nature of Messaging Systems

Senders and receivers remain anonymous to each other since they connect via the provider. The producer and the consumer of messages require the provider to specify the destination of messages so that the producer can send messages to this destination and the consumer can collect messages from this destination

Reliable Delivery of Messages

The sender and the receiver do not have to be available at the same time. The provider delivers the message when the client becomes available.

Messaging Domains

There are two kinds of messaging domains:

- Point-to-Point (PTP)
- Publish-Subscribe(Pub/Sub)

In PTP domains, messages are sent to a particular destination where they queue. A client application is delivers messages from this queue to the destination specified by the provider. Though there can be several messages in the queue, each messages is intended for only one destination/receiver. The Pub/Sub domain, on the other hand, allows a message to be distributed to more than one subscriber via the provider.

Both these domains can be deployed by FioranoMQ. In addition, FioranoMQ can handle the unified domains introduced by JMS 1.1.

Mode of Consumption of Messages

A message can be consumed synchronously or asynchronously.

In the synchronous mode, the client application requests the next message. The client can receive the message in more than one way.

In the asynchronous mode, the client application identifies a message listener. Whenever a message arrives for the destination defined, the provider delivers the message to the subscriber by invoking the `OnMessage` method.

Administered Objects

JMS providers can employ different methods of delivering messages. To make JMS clients portable proprietary parts are encapsulated within JMS objects and are created by the message provider's administrator. These objects are stored in JNDI namespace and can be used by clients through JMS interfaces.

There are two types of JMS administered objects:

- **ConnectionFactory** – is the object used by a client to connect with a provider
- **Destination** – is the object used by a client to specify the destination from which it is receiving messages and to which it is sending messages.

Sessions

A session is the single-thread context for producing and consuming messages. It can create and serve multiple producers and consumers.

A session can be either transacted or non-transacted. Each session supports a single series of transactions and treats them as a unit. Messages produced and consumed within a transaction become the content of that particular transaction. A `commit` method indicates that message processing can occur. A `rollback` method disables the processing of messages. In both cases a transaction is considered to have been completed. A non-transacted session receives message in a mode specified by JMS 1.1: This mode could be one of the following modes: `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK`. The `DUPS_OK` is used in applications where messages delivery can be duplicated.

JMS Message

A JMS message consists of a header property which allows adding an optional header fields in a message and a body. There are five kinds of messages: StreamMessage, MapMessage, TextMessage, BytesMessage, and ObjectMessage. FioranoMQ extends the TextMessage by providing an additional message type: XMLMessage.

Salient Features of FioranoMQ

High Availability

Financial systems that require near-zero downtime require high availability solutions. FioranoMQ HA deployment allows JMS clients to switch to a secondary MQ Server upon failure of a Primary Server.

Client applications have the capability to store and forward messages with automatic re-connection to the backup server. In the event of fault or failure, entire information in the primary server is made available on the backup server so that applications can continue to access this information. This provides applications with automatic fault-tolerance capabilities.

Clustering

Clustering consists of multiple server instances, running concurrently, to provide increased scalability and reliability. The clustering function enables clients connected on different FioranoMQ Servers to exchange information without limiting one client to connect to server at any given point of time. FioranoMQ provides clustering support with the help of three components: the Dispatcher, the Repeater, and the Bridge.

Fiorano's load balancing architecture involves the use of a Dispatcher-enabled server, to route the incoming client connections to the least loaded server within a cluster. The dispatcher component is connected to multiple FioranoMQ Servers. All these servers become part of the cluster that is served by the dispatcher. The repeater and the bridge components are used for Server-to-Server Communication.

XA Support

Real-world applications require transactions involving multiple resource managers. Such transactions are known as distributed or global transactions. Implementation of distributed transactions involves following the JTA standards. FioranoMQ supports both local and global transactions. If a global transaction is active, all activities performed become part of this transaction, or else they operate locally.

Scalability

The load balancing and failover protection architecture allows high scalability in terms of the number of concurrent client connections allowed by a FioranoMQ Server.

Application Server Integration

FioranoMQ integrates seamlessly with several popular J2EE application servers including, WebsphereMQ, JBoss, and WebLogic among others.

Native Runtime Support

FioranoMQ includes client libraries written in C, C++ and C#. These native runtime libraries allow non-java clients to talk directly to the java server as well as exchange information with other JMS clients.

Security

The security implementation includes integrated JSSE support. The Java Secure Socket Extension (JSSE) enables secure internet communications. It implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Developers can, therefore, provide secure channels for data transfer between clients and servers.

Durable Connections

Durable Connection support provides client applications with a fault-tolerant connection mechanism. If an application creates a durable connection, it needs not to worry about re-connecting back to the server in case of fault or failure. This is automatically handled by FioranoMQ's runtime library. If a message is sent during the disconnected phase, it is stored in a local repository of the client machine.

Large Message Support

FioranoMQ enables applications to transfer large messages employing the point-to-point model or the publish-subscribe model. The implementation takes care of resuming data transfer from the point of failure, if any, during the transmission process.

Hierarchical Topics

Destinations with hierarchical dotted naming convention are supported. For instance, a Topic or Queue can be named as `enterprise.finance.admin` or `enterprise.finance.analysts`. Users can then address multiple destinations using wildcards after a '.' in the hierarchy. Topics can thus be broken down into parent and child hierarchies.

HTTP Support

To provide users secure access, FioranoMQ provides Hypertext Transfer Protocol (HTTP) over Secure Sockets Layer (SSL).

Logging Facilities

FioranoMQ has tracing and logging facilities for easy detection of location of errors in the messaging system. The FioranoMQ Administrator has the option of setting different tracing levels for each individual FioranoMQ component.

Message Snooping

Administrators can view messages published on topics as well as on queues. The ability to snoop about/around messages allows the administration, management, testing, and debugging of JMS applications.

Dead Message Queue

A message can be associated with a timeout period within which it is to be received. A dead message queue stores messages that have timed out.

Encryption, Compression Support

Encryption and compression support can be applied for a single message or for all messages to be sent to a particular destination. The default encryption of a message is based on DES. The default implementation of compression is based on Zlib implementation.

Note: In addition message browsing is provided.

Samples

FioranoMQ comes bundled with sample applications that display its features. Experimenting with these applications gives the user a working perspective of the product.

Chapter 2: Configuration Concepts

This section serves as an introduction to the configuration and component model used by FioranoMQ.

FioranoMQ's component model:

- Deployment Profile
- Configuration Modes
- Off-line
- On-line
- JMX
- API's

This chapter explains FioranoMQ's component model. It also introduces the reader to the "Deployment Profile" while explaining the profiles pre-bundled with the FioranoMQ installer. The user is then shown how to choose the right options for configuring the server.

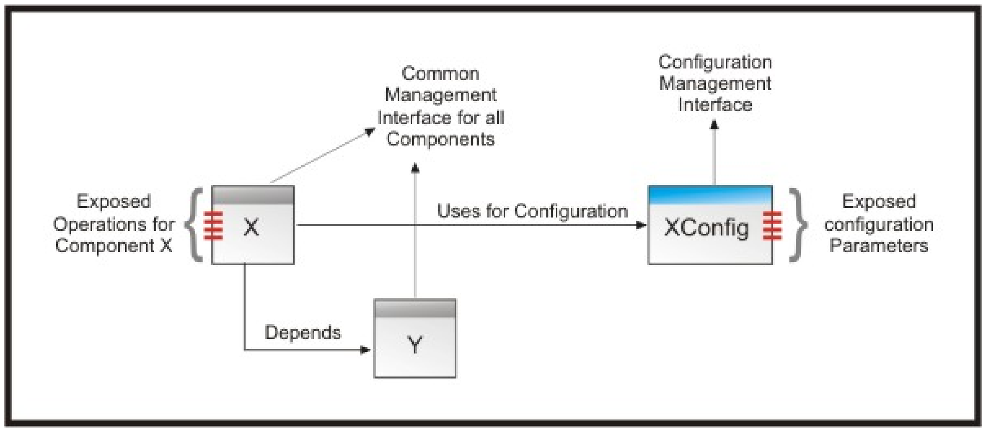
Fiorano Component Model

The FioranoMQ Server implements a componentized model for various internal modules. Components can be clubbed together to form a deployment profile. Each deployment profile can be separately configured both in offline and online mode.

The FioranoMQ Server comprises of a number of components that implement functions independently.

A component:

- Has a well-defined interface through which its life-cycle is controlled.
- Is associated with a unique profile configuration object that defines its configuration needs.
- Can expose configuration attributes and/or operations to the external world via JMX.
- Defines dependencies upon other components.

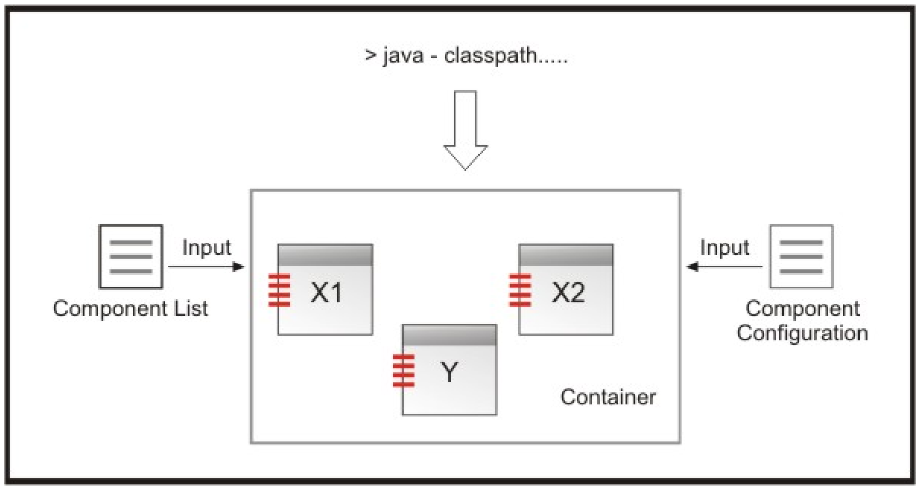

Common Management Interface for all Components
Configuration Management Interface
Exposed Operations for Component X
X
Uses for Configuration
XConfig
Exposed configuration Parameters
Depends
Y

A component is not a standalone executable application but can be hosted only within a Container.

A container:

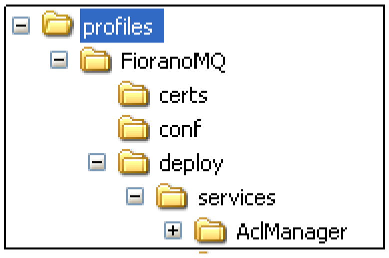
- Is an executable Java program.
- Requires a list of components as input.
- Resolves component dependencies by launching components in the correct order.
- Provides access to a component's configuration upon its launch.
- Encapsulates a JMX MBean Server to which all deployed components are bound.

Allows the invocation of operations and displays changes in exposed configuration parameters of a component.



Deployment Profile

A component only provides a set of services without knowing much about its surroundings. A component is not aware of the application that it encapsulates. Similarly a container is unaware of the content of the "application". The container views components working together in harmony without value judgments. An "application" is defined via the list of components (`deployment.lst`) accepted as input by the container. Modifying this list modifies the behavior of the application hosted in the container. This list along with the configuration for components makes up a "deployment profile". In other words, a deployment profile consists of a list of files (meant for deployment and configuration) organized in a pre-defined directory structure. A typical profile structure (on which the server hasn't been run) is shown below:



Directory	**Description**
certs	Certificate indicating that the server is running in SSL.
conf	Database Configuration Files (*.properties and *.cfg). Confix.xml (Signifying configuration of components).
deploy	*.lst files –comprises the list of components to be deployed in the container when this profile is in use. .
deploy/services	XML files defining dependencies and object names.
run	Directory created when profile is run initially. This becomes the default file-based database storage location of a profile.

In other words, the FioranoMQ Server is an "application" that runs within a Container. The FioranoMQ Server consists of a number of components that provide varied functions such as: pubsub, ptp, admin, and so on. Modifying the deployment profile can alter the behavior of the server. It is possible to run a bridge or a repeater component along with the FioranoMQ Server within the same JVM. This is made possible by the compartmentalization of the components of bridge and repeater. FioranoMQ features such as the XA, can be turned Off or On by removing the corresponding components from the deployment profile.

Each Profile can be thought of as a separate and independent 'work-areas'. The data store and logs stored in the "run" folder are created when the server is launched for the first time.

Default Profiles

The FioranoMQ Server is installed with pre-created profiles that are configured for certain server functions. These profiles are located in "profiles" directory of the server and are summarized in the table below:

Profile	**Description**
FioranoMQ	Default FioranoMQ profile
FioranoMQ_ClusterManager	Pre-configured profile for running fmq server with Cluster Manager enabled
FioranoMQ_Dispatcher	Pre-configured profile for running fmq server with Dispatcher enabled
FioranoMQ_HA_rpl/HAPrimary FioranoMQ_HA_rpl/HASecondary	Pre-configured profile for running primary fmq server with HA (replication) enabled Pre-configured profile for running secondary fmq server with HA (replication) enabled
FioranoMQ_HA_shared/HAPrimary FioranoMQ_HA_shared/HASecondary	Pre-configured profile for running primary fmq server with HA (shared) enabled Pre-configured profile for running secondary fmq server with HA (shared) enabled
FioranoMQ_XA	Pre-configured profile for running fmq server with XA enabled
StandAloneBridge	Pre-configured profile for running bridge on an independent JVM
StandAloneRepeater	Pre-configured profile for running repeater on an independent JVM

By default, the Container runs on the 'FioranoMQ' profile. To choose another profile for the container, specify the profile name along with the `-fmq.profile` parameter in the command line when launching the container.

For example, in order to use "FioranoMQ_XA" profile, launch the container using the command: `fmq.bat -fmq.profile FioranoMQ_XA`

Configuration Tools

FioranoMQ can be configured in several ways:

- FioranoMQ provides a graphical tool, called Fiorano Studio, to configure and manage one or more FioranoMQ Servers.
- Configuration information can be accessed and modified using Fiorano Studio in an offline mode if the server is not available.
- Fiorano Studio allows a user to manage a FioranoMQ server in ‘run’ mode through online configuration.
- FioranoMQ provides comprehensive support for JMX. Any standard JMX-compliant tool can be used to administer FioranoMQ server. Fiorano ships a JMX-compliant administrative tool, called Fiorano JMX explorer, along with FioranoMQ.
- Administration is undertaken using the FioranoMQ proprietary administration API.

Chapter 3: Connection Management

FioranoMQ Server modules include a transport layer which interface with underlying network protocols and accepts incoming connections. This layer reads requests sent by applications over the network. .

The responsibilities of the transport layer include:

- Listening for incoming client connections on pre-defined protocols (HTTP/TCP).
- Management of threads.
- Analyzing incoming data and passing it onto core FioranoMQ services.
- Detecting any loss of connectivity and taking remedial actions.

Socket Acceptors

Socket Acceptors represent input ports through which the server monitors incoming connections. Each socket acceptor is associated with a port number, a transport protocol, a connection manager, and an optional security parameter.

Port Number

Refers to the physical TCP/IP port through which the server monitors incoming connections. Once a connection is established, the socket acceptor handles all requests coming from the client application. This includes JMS, Admin, Lookup, and internal asynchronous requests coming from the FioranoMQ runtime library.

Note: Since a port cannot be shared between two applications, the port number used by the FioranoMQ Server is unique to its server instance. If two instances of the FioranoMQ Server are to run simultaneously on a machine, then both servers listen on different ports. JMX Requests reach the server via the plugged-in JMX Connector, independent of the SocketAcceptor being used.

Protocol

Protocol refers to the physical transport required by a client to connect to the server. By default, the server is configured to use TCP with the option of using HTTP. SSL can also be enabled over TCP and HTTP.

Thread Management

FioranoMQ Server offers different thread management schemes that differ in their handling of new connections. The default thread management scheme associated with a socket acceptor is configured in a manner to create a new thread for each connection to the server. Other schemes allow the configuration of a fixed-size thread pool to service requests from all the connections.

Security Parameters

The socket acceptor can be configured to enable security via SSL. This can be done both on TCP as well as on HTTP protocols. FioranoMQ provides the implementation of SSL over TCP via Sun's JSSE.

Configuration

By default, the FioranoMQ Server is configured for one Socket Acceptor. This socket acceptor is configured to listen through port 1856 using the TCP protocol. The FioranoMQ Administrator has the following privileges with respect to socket acceptors:

- Can edit the default socket acceptor configuration.
- Can create additional socket acceptor(s).

Note: An additional Socket Acceptor in the server which opens another port for communication over the specified protocol.

Connection Factory

As per JMS specifications, an application uses a connection factory to fetch the details of a Connection instance to connect to the server. The connection factory instance encapsulates all the parameters (like URL, protocol, and so on) required to connect to the server. These parameters are configured to use the default socket acceptor settings and must be modified if the server uses a socket acceptor with a non-default configuration. The server creates the default connection to **factories** when it is launched for the first time. These connection factories are automatically created based on the configuration of the socket acceptor being used.

Note: If multiple Socket Acceptors are used, the default connection factories use the parameters of any one Acceptor. The server can be forced to re-create default connection factories at any point. Connection Factory Configuration Parameters:

Parameter Name	**Description**	**Default Value**
ConnectURL	The server URL format. **Note**: The protocol to be used is not part of the URL.	http://localhost:1856
BackupURLs	Semi-colon separates lists of URLs that should be tried when creating connections (or when revalidating a connection) in those cases where a connection with the server specified in the connectURL cannot be established (fails).	

IsForLPC		
isConnectURLUpdationAllowed	If set to true, the connection factory will record the IP address and port of the server through which it is looked up. This is useful for machines where IP address or port is changed. **Note**: This flag is turned ON for default factory connection.	False
ConnectionClientID	If not null, represents a client ID that is automatically set on a connection created through this connection factory.	Null
PingDisabled	If set to true, a connection created will not be pinged even if pinging is turned ON at the server.	False

Obtaining a Connection Factory Instance

A connection factory is a stateless object that encapsulates information on how to connect to the server.

JNDI Lookup

A connection factory instance is a serializable object that can be stored and later looked up through any JNDI-compliant directory server. FioranoMQ provides the JNDI interface to lookup all admin objects.

Creating a new instance

An application can create a new instance of connection factory and use it after setting various configurable parameters.

Lookup

FioranoMQ applications can lookup objects (destinations and connection factories) when using the JNDI interface. A single socket acceptor can service lookup requests as well as JMS requests. Therefore, in order to send the request to the server, its connection parameters have to be specified as environment variables to JNDI. Server URL, Transport protocol, and security parameters can be specified in the environment forwarded to the JNDI layer.

JMX

FioranoMQ 8.0 and later versions provide extensive support for JMX. This allows any third party JMX-compliant applications to connect to the server remotely and access/modify the configuration at runtime. This requires a JMX Connector to be plugged into service incoming JMX requests. There are two options for JMX Connectors:

1. RMI Connector
2. JMS Connector

RMI Connector

Is the default connector shipped with the FioranoMQ Server. It uses RMI as the underlying transport protocol to establish communication with the server and with a JMX-compliant application. This connector uses a dedicated socket that accepts connections and services requests. By default, the Connector is configured for port 1858 but can be configured to work with another port.

JMS Connector

This connector uses JMS Bus for establishing communication with the JMX Application and with FioranoMQ Server. All connection parameters are treated as configurable parameters. The JMS connector is configured to connect to the FioranoMQ Server running with the default socket acceptor configuration. If this configuration is modified, corresponding changes must be made to the JMS Connector configuration.

Pinging

The transport layer detects loss of connectivity between an application and the server and performs the necessary server cleanup. Loss of connectivity is detected via a ping mechanism between the Server and the Client. By default Pinging is disabled. When enabled, it instructs the FioranoMQ client library to send ping packets periodically to the FioranoMQ Server. The server on its part monitors ping requests on all connections and if it does not receive ping packets for any connection within a configured timeout, it assumes the connection is dead and shuts it down.

When to Enable Pinging

In some operating systems, the absence of any activity over a client socket for a period of time leads to its forceful shutdown by the OS. Enabling Pinging avoids this shutdown.

A network failure can not be detected for applications not sending requests to the server. With Pinging enabled, the application is notified of a network failure prior to the configured timeout.

Salient Features

The Configurable Parameter for Pinging includes the Ping Timeout Interval. This parameter specifies the time within which the client application is notified of a problem with connectivity. By default, this parameter is set to 4 minutes or 240,000 milliseconds. The lowest allowed value for this parameter is 30,000 milliseconds.

Pinging is automatically turned ON when the Socket Acceptor uses the HTTP protocol.

Since a connectivity problem is detected asynchronously for the application, an error is relayed through an exception set on the connection as per JMS Specifications.

Note - In other words, it is mandate to set an exception listener on the connection if the application is to be notified of connectivity problems.

Chapter 4: HTTP Support

FioranoMQ supports HTTP as the transport protocol between JMS clients and MQ Servers. This allows JMS communications to flow through corporate firewalls/proxies. Implementing this feature insulates the protocol layer from JMS application development. The client side environment is responsible for determining the protocol to be used for communication. All protocols - TCP, Secure TCP (JSSE), and HTTP behave in a similar manner. Selecting the protocol to be used is based on the configuration of the application developer. Synchronous and asynchronous communications are available regardless of protocol choice.

HTTP is typically used to communicate across the internet. If the FioranoMQ Server is to directly process messages received from clients over the internet, it must be deployed in a manner similar to a web-server. The HTTP support in FioranoMQ provides the necessary features that make it function as a web server enabling it to handle HTTP requests. Using HTTP Tunneling a direct connection between client and server can be established.

Client Side Changes

While switching protocol from TCP to HTTP, the following changes are required:

- All additional parameters need to be marked as JNDI.
- If `jndi.properties` file is used to specify parameters, the application code need not be modified. In such ases, an HTTP Enabled Connection factory needs to be used.
- Including of `HTTPClient.zip` file in classpath.

Using Proxies

HTTP support of FioranoMQ provides seamless communication via proxy servers. Most proxy servers including- Microsoft ISA server, Wingate, and WinProxy among others – can be used to connect to the FioranoMQ Server. FioranoMQ Client libraries allow developers to set the Proxy Address and Port in the client applications. These can also be set as Java VM Properties.

Proxy Authentication

FioranoMQ supports both "Basic" and "Digest" authentication for communication through proxies. Various Proxy Authentication parameters such as the Authentication Realm, username and password can be specified through client applications using the environment variables.

Authentication is required within the instance of a VM. FioranoMQ caches the authentication information and uses it for other connections.

Tunneling through Firewalls

This section discusses how JMS Clients operate in networks where firewalls are present. FioranoMQ allows enterprise clients to extend beyond corporate firewalls by providing both HTTP Tunneling and tunneling through SOCKS enabled Proxy servers. FioranoMQ provides Tunneling support for clients along with all JMS functionalities.

Tunneling through SOCKS Proxy Server

Tunneling through client as well as server side firewalls can be achieved through the SOCKS Proxy Server. The SOCKS protocol is an open internet standard for performing network proxying at the transport layer. SOCKS creates proxy, which serves as a data channel between TCP or UDP (User Datagram Protocol) based clients and servers. The proxy between the client and server, created by SOCKS is transparent to both the parties.

Java runtime 1.1.8 and above provide SOCKS support. The Java.net socket instance has the ability to connect to a remote host through the SOCKS proxy server. If the System property `socksProxyHost` and optionally `socksProxyPort` is set, the Socket implementation redirects the connection through the SOCKS proxy Server. Tunneling through proxies, using SOCKS, presents a more generic and viable solution for JMS Applets. Since `socksProxyPort` and `socksProxyHost` are set as a system property, the Client Applet burrows through the SOCKS server. A single version of an applet can now be downloaded by the client, despite the presence of a firewall. There are slight variations in the applet and application code used to tunnel through the SOCKS Proxy. Using HTTP Tunneling requires that the applet sets the proxy Address and proxy Port. The code snippets provided in this document illustrate proxy tunneling in applications and applets.

The above features do not work with JDK versions below 1.4 and 1.5. Complete samples can be found in the Tunneling Samples folder located at: `%FMQ_DIR%\fmq\samples\` directory.

Enabling JMS Applets to Tunnel through SOCKS Proxy Server

Browsers allow users to manually set the Proxy Server/SOCKS Server Host and port or users can use a script to automatically set the browser configuration. Applets access Java for SOCKS proxy server settings by conveying the settings effectively to the Java VM, used by the browser.

Microsoft Internet Explorer 4.0 and above provide complete SOCKS proxy support. They do not require changes to run Applets behind client firewalls.

Note: Netscape Communicator does not convey its proxy server settings to Java VM. This can be achieved by using digital certificates. A digital certificate allows the client Applet to set System properties for Java VM. (For more information, refer to the SockPubSub samples directory in the FioranoMQ installation directory.)

Additional Notes on SOCKS

JDK implements SOCKS Version 4. SOCKS Version 4 accepts remote host addresses in numeric IP form (and not alphanumeric form which would allow the use domain names such as www.fiorano.com). Tunneling does not work if issues of domain name and IP address are not resolved. To resolve the issue the Applet needs to be downloaded from a known IP address and used instead of domain names. Another solution is to provide the Server IP Address as Applet parameters.

HTTP Pinging

To get the server to detect the clients that have been disconnected from the server pinging needs to be implemented over HTTP as well. This enables a 'clean up' of resources used by clients that have been disconnected or timed out. To implement this feature, the client library must continuously ping the server at predefined intervals. The server can be configured to perform this action. Pinging is essential for the HTTP support of FioranoMQ and is set as the default feature in the case of HTTP connections. Due to attributes of HTTP, there are limitations in detecting control-C (or terminating the application abruptly) related issues from the client. It is expected that the JMS client application programmers perform an explicit `connection.close()` operation to enable the server to detect client disconnections. Not doing an explicit close can result in inconsistent message reception and delivery.

Chapter 5: FioranoMQ Security

FioranoMQ provides a comprehensive security model. The key benefits of FioranoMQ are:

- Complete implementation of Java 2 Security APIs.
- Design and implementation of JMS Applications, independent of security policy.
- Configuration of security and user privileges through a central administration tool.

To use the internet as a platform for business applications, organizations need to protect access to corporate assets, such as databases. Each class of users (such as employees, customers, partners, and suppliers) requires different levels of access.

Information can travel to many locations over the network; yet confidential messages must remain private. This is achieved through:

- **Authentication**: Determination of user identity
- **Authorization**: Definition and control of user activity

User Identification and Authentication

The FioranoMQ security subsystem provides user identification and authentication using standard JMS APIs. The integrity and privacy of data (discussed in the next section) is protected using MD5 (Message Digest 5) checksums and 40-bit and 128-bit encryption. FioranoMQ supports destination-based security which allows altering access permissions for Topics and Queues stored on the FioranoMQ Server.

To implement the username/password model specified by the JMS API, set up users following the instruction below:

- Set up users through FioranoMQ Administration API/GUI tools. Usernames are stored in the FioranoMQ offline database, together with their passwords and descriptions.
- When a client application tries to connect to the FioranoMQ server using the API `TopicConnectionFactory.createTopicConnection (String username, String passwd)`, the FioranoMQ runtime library (embedded within the client) sends a connection request to the server, with the username and password. The server searches for the username in its repository. If the username is found, the server compares the supplied password with the existing password in the repository. If the password matches, the connection request is accepted, otherwise it is rejected and the client throws an exception.

If the username sent cannot be found in the repository, the server rejects the connection. A valid connection is allowed if the anonymous user is present in the users list. (An anonymous user is shipped with the product.) Additionally, any user can create a connection using the following:

```
TopicConnectionFactory.createTopicConnection(null,null);

TopicConnectionFactory.createTopicConnection("anystring",null);

TopicConnectionFactory.createTopicConnection(null,"anystring");
```

All these connections are equivalent to:

```
TopicConnectionFactory.createTopicConnection ("anonymous","anonymous") call.
```

If this option is not required, then the anonymous user should be deleted through the 'Admin' API. These calls do not allow creation of connections. This is true for `createQueueConnection()` and `createConnection()` calls as well.

Data Protection

The secure version of FioranoMQ Server protects the integrity and privacy of all the messages exchanged between the client and the server.

The integrity feature verifies that message content on delivery matches the original published form. Corruption of data can be accidental or intentional. FioranoMQ uses the cryptographic checksum Message Digest 5 (MD5) algorithm to validate the message content integrity.

FioranoMQ ensures the privacy of a message by using encryption. Encryption scrambles the message content before sending it over the network and restores the original form on delivery. If the message is intercepted before delivery (if someone attempts to read it as it travels over the network), the details are available in an unreadable form. By default, FioranoMQ encrypts messages to provide privacy using a 40-bit encryption provided by the Data Encryption Standard (DES) algorithm. FioranoMQ allows customization of the cipher suite. For example, it is possible to switch from DES 40-bit encryption to 128-bit RSA encryption (domestic version only), or to switch between various available 40-bit encryption algorithms, by setting the SSL parameters.

FioranoMQ 7.0 onwards provides seamless integration with NT realms. The need for the Enterprise Administrator to set up separate user realms for MQ is made obsolete. FioranoMQ integrate seamlessly with existing NT/Solaris realms.

Authentication Based on Digital Certificates

Besides username/password authentication, FioranoMQ incorporates authentication based on digital certificates. This feature is available only on the secure FioranoMQ Server. When certificate based authentication is enabled, each client passes on a one way encrypted version of its digital certificate to the server while trying to establish a connection. The server authenticates the client certificate and, if successful, passes back its own certificate to the client process, allowing the client to verify the identity of the server.

Security Realms

FioranoMQ supports Realm based security that allows FioranoMQ to integrate with Solaris and NT Security realms. This eliminates the need to create MQ specific users/permissions.

A realm is an administrative entity around which basic operational security policies revolve. A realm determines the scope of the security data and is normally used to organize the objects used in defining access control policies.

Security realms represent a logical grouping of Users, Groups, and Access Control Lists (ACLs) for protecting FioranoMQ Server resources. The default security realm or one of the sets of alternative security realms can be used, which allow usage of Windows NT, UNIX, and LDAP (Lightweight Directory Access Protocol) security stores. In addition, FioranoMQ supports custom developed security realms.

A Realm object provides access to users and the main Principals around which a realm is organized, and supports modifying (and extending) it according to policies defined by the realm administrator and by each particular kind of realm. Different Realms use different Authentication Protocols such as passwords (or pass phrases) and public key certificates. Groups of users (and of other groups) are used to define various policies applying to many users. ACLs are uniquely associated with entries in each realm.

FioranoMQ implements a sophisticated security engine that allows dynamic updating of Users/Groups and their privileges. Users, Groups, and ACLs can be retrieved as needed from an external source. FioranoMQ Realms Subsystem is divided into two services: User Management and Access Control Management, each of which is discussed in the following sections.

FioranoMQ User Management

FioranoMQ User Management service uses Realms to retrieve Users and Groups as Java objects.
Any one of the following realms can be chosen for User Management:

- Default Realm
- NT Realm
- RDBMS Realm
- LDAP Realm
- Caching Realm
- XML Realm

The User Manager implementation can be specified in the profile deployed during configuration.

Access Control Management

FioranoMQ includes a powerful and flexible access control system to control access to applications and to backend services that clients access through the FioranoMQ Server. The access control system is built on the Java2 security APIs.

An ACL guards an object or service in the FioranoMQ Server. ACLs can guard Topics and Queues. Additionally, custom ACLs can be created for use in applications. An ACL holds a list of ACL entries, each with a set of permissions for a user or group. Permission is actions that can be performed on the protected destination, for example, publish, lookup, and subscribe.

FioranoMQ's dynamic verification engine is invoked before any service call is executed, which checks pertinent ACLs, testing whether the user has the permission required to continue.

By default, FioranoMQ uses the file-based data store for storing ACL information. ACLs are associated with realms in such a way that the entries in them, which identify users and groups, are only significant within a particular realm. FioranoMQ realms are dynamic; they retrieve Users, Groups, and ACLs as needed from an external source.

More information about Access Control Lists is available in the Java documentation of the java.security.acl package.

Any of the following realms can be chosen for ACL management:

- Default Realm
- RDBMS Realm
- LDAP Realm
- XML Realm

The ACL Manager Implementation can be specified in the profile deployed during configuration.

Default Realm

The default security realm is the File-based Security Storage System in which the User/Group information and credentials are stored in the file based persistent store.

NT Realm

Using Fiorano NTRealm avoids defining Users and Groups on FioranoMQ. Windows NT Security realm of FioranoMQ uses account information defined for a Windows NT domain, to authenticate Users and Groups. FioranoMQ NTRealm provides authentication using the WindowsNT security domain controller.

Salient Features

Fiorano NT Realm requires the FioranoMQ Server to be run as a Windows administrative user, enabling it to read security-related data from the Windows NT Domain Controller. To use Fiorano NT Realm, FioranoMQ must be run on a computer in the Windows NT domain.

To manage User and Group information, the FioranoMQ Server must be able to make system calls on the Windows NT computer, where the FioranoMQ Server is running. In other words, FioranoMQ needs appropriate privileges to be able to communicate with the Primary Domain Controller to perform authentication.

NT Principal Manager, only users registered in Administrators group has rights to open/create Admin Connection. Other users can be given these rights by adding/registering them to the default Administrators group.

Note: User admin (used by default to create admin connections) is not a member of the Administrators Group in FioranoMQ NT Realm. To use FioranoMQ default admin tools and APIs, the admin user must be registered in the Administrators group.

Limitations

- A Group can not have a Group as a member.
- Password for a user can not be changed using Fiorano NT Realm API. A password can be changed using the Windows NT Administration Tool, instead.

Note: These limitations relate to the NT implementation of `realm.principal` and can be over-ridden by using any other implementation of FioranoMQ Realm.

Troubleshooting

The most common configuration problem within Fiorano NT Realm is related to Windows NT policies and, specifically, with the user account that runs the FioranoMQ Server. The user account that runs FioranoMQ Server requires special permissions to access the Windows NT domain. The steps for granting these permissions are available through the configuration instructions.

A frequently occurring problem concerns FioranoMQ Server's difficulty in loading the file `fioranorealm.dll`. If FioranoMQ is unable to load the `fioranorealm.dll`, it gives the following message:

```
java.lang.UnsatisfiedLinkError: no fioranorealm in java.library.path

at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1312)

at java.lang.Runtime.loadLibrary0(Runtime.java:749)

at java.lang.System.loadLibrary(System.java:820)

at fiorano.jms.realm.principal.nt.FioranoNTManager.init(FioranoNTManager.java:82)

at fiorano.jms.realm.principal.nt.FioranoNTManager.(FioranoNTManager.java:51)

at
fiorano.jms.realm.principal.nt.PrincipalManagerImpl.startup(PrincipalManagerImpl.java:
61)

at fiorano.jms.realm.RealmManagerImpl.startup(RealmManagerImpl.java:77)

at fiorano.jms.ex.Executive.startup(Executive.java:647)

at fiorano.jms.ex.Kernel.startup(Kernel.java:61)

at fiorano.jms.ex.fmpmain.main(fmpmain.java:60)

fiorano.jms.common.FioranoException: REALM_NOT_SUPPORTED :: NT realm support is not
available
```

RDBMS Realm

The RDBMS security realm is a custom realm that stores Users, Groups and ACLs in a relational database. It uses configuration information to obtain database connection information. Using the connection information it connects to the database and loads Users, Groups, Permissions, and ACLs. Since the methods for loading and saving the Realm modify the values, the Realm is maintained in a 'stored state' rather than saved. Configuring the RDBMS Security realm involves setting fields that define the JDBC driver used to connect to the database. Additionally, it defines the schema used to store Users, Groups, and ACLs in the database.

Directory	**Description**
DbDriver	The full class name of the JDBC driver. This class name must be in the CLASSPATH of FioranoMQ Server.
URL	The URL for the database used with the RDBMS realm, as specified in the JDBC driver documentation.
UserName	The username of the database user.
Password	The password for the username.

LDAP Realm

LDAP Realm provides authentication using the Lightweight Directory Access Protocol (LDAP) server. This enables the management of Users, Groups and ACLs from one location, the LDAP directory. LDAP realms allow storage or usage of ACL/user information on any external LDAP server. When the LDAP security realm is used, the LDAP server authenticates Users and Groups.

In the case of SSL protocol (with FioranoMQ Server), the LDAP Security Realm retrieves a common name of the User from its digital certificate and searches the LDAP directory for that name. The LDAP Security Realm does not verify the digital certificate. This verification is performed by the SSL protocol. The LDAP Security Realm currently supports Netscape Directory Server, Microsoft Site Server, OpenLDAP, and Novell NDS.

Configuring the LDAP Security Realm

Configuring the LDAP Security realm involves defining the fields that enable the LDAP Security realm, within the FioranoMQ Server, to communicate with the LDAP server. Additionally, it involves defining the fields that describe how Users and Groups are stored in the LDAP directory. These fields are described in the Table.

Directory	**Description**
LdapProviderURL	Location of URL server. Change the URL to the name of the computer on which the LDAP server is running and to the port number at which it is listening. If the FioranoMQ server needs to connect to the LDAP server using the SSL protocol, the LDAP server's SSL port in the URL should be used.
Principal	The distinguished name (DN) of the LDAP User is used by the FioranoMQ server to connect to the LDAP server. The User must be able to list the LDAP Users and Group.
Credential	The password that authenticates the LDAP User, as defined in the principal field.
LdapsecurityAuthentication	Determines the method for authenticating Users.
LdapUserPasswordAttribute	Password of the LDAP User.
LdapUserDN	A list of attributes which combined with attributes in the username attribute field uniquely identify a LDAP user.
LdapUserNameAttribute	The loginname of the LDAP User. The value of this field can be the common name of an LDAP user, but usually it is an abbreviated string, such as User ID.
LdapGroupDN	A list of attributes which combined with the Group name attribute field uniquely identifies a Group in the LDAP directory.
LdapGroupNameAttribute	The name of a Group in the LDAP directory. It is usually a common name.
LdapGroupUsernameAttribute	Name of the LDAP attribute that contains a Group member in a Group entry.

Miscellaneous Features

If caching is enabled, the Caching Realm internally caches Users and Groups to avoid frequent lookups to the LDAP directory. Each object in the Users and Groups cache has a TTL field (TimeToLive), which is set while configuring the Caching realm. If changes are made in the LDAP directory, those changes are not reflected in the LDAP Security realm until the cached object expires or is flushed from the cache. The default TTL is 60 seconds for unsuccessful lookups and 10 seconds for successful visits. Changes in the LDAP directory should be reflected in the LDAP Security realm within 60 seconds, unless the TTL fields for User Groups and caches have been changed.

If server-side code has performed a lookup of the LDAP Security realm, such as a getUser() call on the LDAP Security realm, the object returned by the realm cannot be released until it is released by the code. Therefore, Users authenticated by FioranoMQ Server remain valid as long as the connection persists, whether or not the User is deleted from the LDAP directory.

Schema checking is turned on by default in the directory server, and Netscape recommends running the directory server with schema checking turned on. The schema checking is turned off for realmLDAP.

XML Realm

FioranoMQ provides an XML based Security Storage System in which User/Group Information, Credentials of Users and Groups, as well as ACL information is stored in an XML format.

Caching Realm

The Caching Realms works with File Alternate Security or Custom Security Realms to fulfill client requests, given proper authentication and authorization. The Caching realm stores the results of both successful and unsuccessful realm lookups. It manages a separate cache for Users, Groups and authentication requests. The Caching realm improves the performance of the FioranoMQ Server by caching lookups and reducing the number of calls to other Security Realms.

Caching can be used with any FioranoMQ supported Security Realm. The separate lists of resources, such as users, groups and user-passwords are cached. Caching avoids repeated calls to the underlying security store.

FioranoMQ Security - Salient Features & Advantages

Design Advantages

FioranoMQ allows developers to focus on building the application and not on implementing a security policy. Security operates independent of application code through an easy-to-use, central administration interface that manages Users, Groups, and Access Control Lists (ACLs). This design permits remote administration for all aspects of security. If security policies of an organization change, the system administrator can manipulate security mechanisms of FioranoMQ without requiring the application developers to rewrite any application code. By allowing security policies to change with business needs, FioranoMQ provides the flexibility that extends the life of an application.

Effective Protection of JMS Destinations

FioranoMQ achieves security by protecting the JMS Destination: Topics and Queues. This enables security to be addressed through the design of Topics and Queues Existing applications take advantage of new security features as soon as they become available in newer versions of FioranoMQ.

Centralized Control

The identification and authentication process is the only area where the client application must address security. The application developer is responsible for the task of soliciting and passing the username and password to FioranoMQ. This requires that the application access sufficient information from the User to allow FioranoMQ to authenticate the User. This is the only information the client application code needs for security. The system administrator uses an external Administration Tool to visually set the security policies, which FioranoMQ enforces.

FioranoMQ and its security subsystem do not require any pre-existing software to be installed by the client. All security functions are inbuilt in FioranoMQ and provided through standards-based Java Development Kit (JDK) interfaces. As such, security is built into each and every application or applet that is created by FioranoMQ. Moreover, a developer need not worry about whether the application runs locally or as a downloaded applet. The FioranoMQ security subsystem does not require access to any local (client-side) resources. The security subsystem uses either part of the Java runtime environment or classes downloaded with the applet to function.

Destination-based Security

In FioranoMQ, all the information flow is based on destinations, as explained below:

- Developers organize content based on destinations.
- Applications can register their interest in consuming the information by subscribing to a destination.
- Applications can produce information by publishing messages to destinations.
- The FioranoMQ Server routes information from publishers to subscribers based on the destination.
- The security subsystem takes advantage of the dependency of information flows within destinations. By protecting the destination, the flow of information can be precisely and dynamically controlled. FioranoMQ refers to this as destination-based security. FioranoMQ associates a security policy with every destination.

Authorization and Access Control

FioranoMQ provides the ability to control Users that can publish, subscribe, or request guaranteed delivery on a particular destination, through the use of Access Control Lists (ACLs). The system administrator uses the Administration Console to define ACLs for specific destinations. FioranoMQ automatically uses these ACLs as described in the following section.

The FioranoMQ Server performs access mediation on publish and subscribe operations of a client and on guaranteed delivery requests. For example, when the client subscribes to a destination, the server receives the policy for the destination and checks to verify whether the client is permitted to subscribe to that particular destination. If durable subscription is requested on the destination, an access check is also performed for durable subscription at the time the DurableSubscriber object is created. If either one of these checks fail, the subscription request is rejected and the client application throws an exception.

When a client publishes a message on a destination, the server checks to verify, whether or not the client is authorized to publish on that destination. If the client is not authorized, the publish request is rejected and the client application throws a JMS Exception. If the client is authorized, the server delivers the message to all the clients subscribed to the destination of the message.

Access mediation is performed on the server side. However, a client can check for permission to publish, to subscribe or to request a guaranteed delivery of messages to a specific destination by retrieving the appropriate ACL object and examining its contents.

The system administrator uses the administration console to add new Users and Groups and/or new policies for destinations.

Default Users, Groups and ACLs

By default the following users are created in FioranoMQ:

- Admin
- Anonymous
- Ayrton

Each user is a member of "EVERYONE" group. Depending on the configuration parameter `CreateDefaultAcls` (where the default value is true), the ACLs are created for all the topics and queues at the startup of the server.

Chapter 6: FioranoMQ Data Stores

The messaging system uses a store to save persistent JMS messages.

Storage type

Persistent messages can be stored using:

File based store: FioranoMQ has a proprietary mechanism to store/retrieve messages from a flat-file based store.

OR

RDBMS based store: Messages can be stored in a JDBC-compliant RDBMS. Any standard RDBMS like Oracle, MSSQL, MySql, IBM DB2, Cloudscape, and HSQL can be used to store the messages.

An administrator can configure the server to store messages in these message stores.

By default, FioranoMQ is configured to use a file-based message store. FioranoMQ also provides the support for using different stores for the two messaging domains. There is one store for messages on PTP domain and another for messages in the Pub/Sub domain.

The type of store to be used for messages associated with a particular destination can be specified during the creation of the destination. Administrators can specify the storage type of a destination using:

- The FioranoMQ Administrator Console
- The FioranoMQ Administration API

File-based store versus JDBC-compliant RDBMS store

The performance of FioranoMQ with an RDBMS-based message store is not as optimal as with a file-based message store.

RDBMS based store provides higher security.

Offline Database reliability is generally higher.

Database stores generate network traffic if the database server is on a different JVM or machine. Network traffic is not generated in case of a file-based store.

Default Destinations for sample applications

FioranoMQ creates default destination objects that can be used by client applications to run samples provided with the FioranoMQ installer. By default, FioranoMQ uses the file-based message store.

It is possible to configure the FioranoMQ Server to use the file-based message store or an RDBMS-based message store.

File-based default destination objects are:

- PrimaryQueue
- SecondaryQueue
- PrimaryTopic
- SecondaryTopic

The messages published on these destinations are stored in a file-based message store.

RDBMS-based default destination objects are:

- PrimaryRDBMSQueue
- SecondaryRDBMSQueue
- PrimaryRDBMSTopic
- SecondaryRDBMSTopic

Messages published on these destinations are stored in the RDBMS-based message store.

By enabling both file-based and RDBMS-based stores in a single instance of the FioranoMQ server enables messages requiring fast retrieval to be saved in a file-based store and messages requiring the reliability of JDBC-compliant relational databases to be saved in an RDBMS-based file store.

Creating a Default Database

FioranoMQ can be configured to store the messages in a JDBC-compliant relational database. By default, FioranoMQ is configured to use the HSQL database. Fiorano ships the library containing the JDBC driver for HSQL with the FioranoMQ product. If a different RDBMS is required, administrators need to create the FioranoMQ database in their RDBMS. If FioranoMQ needs to store messages in an Oracle database, administrators are expected to create database tables for storing messages and related content in the Oracle database. FioranoMQ provides scripts that can be used to create the required tables. These scripts need to be executed before running the RDBMS enabled FioranoMQ server. These scripts accept the URL, UserName and Password as system variables and then create the file-based or RDBMS-based databases.

A FioranoMQ database can be created using:

- Command Line parameters
- Fiorano Studio

Clearing a Database

A script is provided for clearing both file-based and RDBMS-based databases.

Chapter 7: Managing Administrated Objects

JMS entities like Destinations and Connection Factories (also collectively known as administered objects) are well defined in JMS Specifications. The roles of these objects as well as their APIs are well defined. However JMS specifications do not state how to obtain an instance of these objects. It is left to the JMS Provider to provide instances of these objects to an application.

FioranoMQ allows applications to use JNDI APIs to obtain instances of Administered objects

Note: For additional information on JNDI, please refer to:
http://java.sun.com/products/jndi/

The use of JNDI allows the application code to be **Standards Based** without the need to import Fiorano specific classes. FioranoMQ provides a limited implementation of the directory server. This is done so that an end user need not configure information from a third party directory server for using FioranoMQ.

FioranoMQ also allows applications to store and retrieve administered objects in a third party external LDAP server. FioranoMQ allows its applications to create a new instance of an administered object (destination or queue) and use it in all JMS operations. This requires using non-standard Fiorano specific APIs in the application.

Naming Services

The Naming Manager, a module within the FioranoMQ Server, provides all common Naming Services (lookup, bind, delete, list, and so on). Naming requests, originating from a JMS Application, are sent to the FioranoMQ server is internally forwarded to this module, which processes the request and responds accordingly. Administrative requests leading to creation or deletion of administered objects are also forwarded to this module.

The interface for this module is well defined and allows multiple implementations which differ mainly in the persistent media used for storing information. A FioranoMQ Administrator is free to plug in any implementation (or even write a new implementation and plug it in) of this interface.

File

This is the default implementation for Naming Manager. It stores information in a proprietary File (defaults to `admin.dat` in run folder of the profile).

XML

This implementation stores information in clear text format in an XML file (`admin.xml` in the run folder of the profile).

LDAP

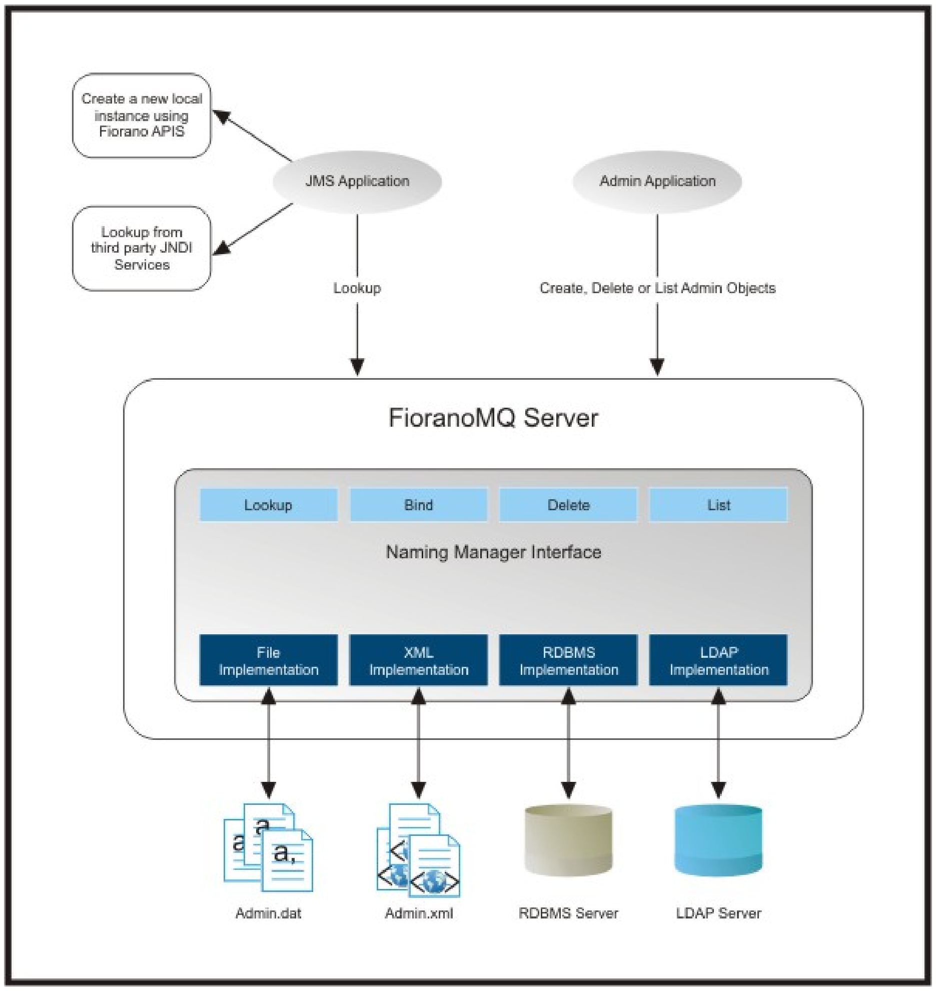
This implementation uses a third party JNDI compliant Naming and Directory Service to persist information.

RDBMS

This implementation uses third party JDBC compliant RDBMS server to persist information.

Cache

This implementation creates a "cache" of admin objects in memory. The cache can be used with any of the above implementations for storage purposes. Caching is beneficial in situations where there aren't frequent changes in the admin object store.



Salient Features

FioranoMQ can be configured to use any of the above Naming Manager implementations. The configuration here takes place off-line.

The JNDI implementation provided by FioranoMQ is limited and provides the implementation of only basic methods. It should not be considered a standalone JNDI implementation.

Chapter 8: Message Expiry

JMS Standards allow setting Time To Live (TTL) on a message being sent to a destination. The JMS provider considers this message valid up to the period specified in the TTL. Once the TTL period elapses, the message expires, ceases to be available on the destination and fails to be delivered.

Point of Checking of Message Expiry

The server adds the current specified time of TTL to the current time and obtains the expiry time. This is done when the message fist enters the server. The server checks for expiry when attempting to deliver it to a consumer. If expired, the message is ignored.

Since expiry is checked just before delivering the message to a consumer; if there is no active consumer, expired messages might continue to consume server resources (disk or memory space). To optimize performance and server resources, the server can be configured to check expired messages in all queues periodically. To enable this, set the value of the flag `DbCleanupEnabled` to true. (By default, it is set to false.) The frequency with which the server checks for expired messages is configured through the parameter `CleanupInterval`. (By default set to 10 minutes.)

On Detection of an Expired Message

Once the server detects an expired message, it deletes this message from the destination. Since this deletion is done automatically the following actions can be performed: A Copy of the message is pushed into the Dead Message Queue.

A Copy of the message is published on an Admin Topic

Note: Publishing on admin topic will be discontinued in future releases. Instead, the server fires a JMX Notification with information about the expired message.

The sections below provide more details on dead message queues and the handling of expired messages.

Dead Message Queue

Dead Message Queue is a special system queue with the name SYSTEM_DEADMESSAGES_QUEUE created for storing copies of messages that expire in any of the server destinations. Any client applications can browse or receive messages from this queue using normal JMS semantics.

DMQ Configuration

Controls are provided to configure DMQ functionality globally for multiple queues as well as individual queues. These controls are:

Parameter	**Scope**	**Possible Values**
`EnableDMQOnAllQueues`	Global	Yes and No
`EnableDMQ`	Individual Queue	Yes No and Default

By default, individual queues have `enableDMQ` set to `True`. This allows the administrator to control DMQ configuration for all queues through global flags.

Other DMQ configuration parameters are summarized in the table below:

Parameter	**Description**
`DMQExpiryTime`	The time period that messages would live on DMQ.
`CleanupDMQAtStartup`	If set to Yes, all DMQ messages would be deleted at server startup.

Selectively disabling DMQ for a message

If DMQ is enabled for a destination, by default all expired messages are added to DMQ. However, if an application doesn't want to use the DMQ functionality, it can do so by setting properties in the message through various APIs.

Message Expired Notifications

When a message has expired, the server (if configured) publishes a notification in the form of a JMS Text message on a system topic named ADMINISTRATOR_TOPIC. This function can be used to get notification of expired messages. Any application can create a subscriber, based on JMS semantics that receive these notifications.

Configuration

Notifications can be configured globally through a flag `EnableNotificationOnDeadMessage`. If this flag is set to `true`, the server publishes a notification when a message expires. If an application wants to disable this function for specific messages, it can do so through APIs.

Additional points

Notifications work only if DMQ is configured for the queue. The published Text Message has the following attributes:

1. It has the same set of properties as the original message that expired.
2. Its body contains (as text) the destination name on which the original message was published. The message is published to the DMQ as a non-persistent message. This feature is discontinued in future releases. New releases fire JMX Notifications when a message expires.

Chapter 9: Snooper

Snooper is a FioranoMQ feature that allows an application or an individual to view incoming messages. This feature is to facilitate debugging of JMS applications.

Snooper functionality can be used through Fiorano Studio (Customized GUI tool). This tool allows the administrator to enable/disable the snooping function and can also show the contents of a "snooped" message in a tabular manner in the GUI. A console-based application can also be written in order to snoop messages. This application can receive the published messages as well as inspect them.

Snooper Configuration

To snoop messages on a destination, the snooping function has to be turned ON for that destination. This can be done through Studio as well as programmatically through a Java Application using Admin APIs.

Besides enabling/disabling this function for a destination, the Snooper configuration on a destination can be left set to "Default". If this configuration is not set, global flags decides whether or not snooping is be turned on for a destination. This provides the flexibility of setting the snooping function on all queues and/or topics at the same time.

Note: The values of global parameters are consulted only when the snooper configuration is set to Default. For other values (on/off) the global parameters are ignored. All FioranoMQ destinations are configured to this value.

Working of Snooper

If snooping is turned on, the FioranoMQ Server sends a copy of the incoming messages to pre-configured system topics. An application can then pick up this message and inspect the same.

The System topics used for snooping are:

- `SYSTEM_MESSAGE_SNOOPER_TOPIC`
- `SYSTEM_MESSAGE_SNOOPER_QUEUE`

A message on a topic is sent first; messages coming on a queue are sent latter.

Security Settings

Security Settings for Snooping are controlled by the ACLs of the above system topics. By default the following restrictions apply:

- Durable subscriptions are not allowed.
- Only the FioranoMQ administrator can "snoop".
- Only FioranoMQ Administrators can edit the ACLs of these topics to modify restrictions. The ACL name is the same as the name of the system topics described above.

Miscellaneous Features

Important points that a user should remember:

- "Snooped" messages are a copy of original messages. Making changes in snooped messages would not affect the actual message.
- "Snooped" messages are always delivered as "Non Persistent" even if the original incoming message was persistent.
- Snooping on system topics is not permitted.

Chapter 10: Durable Connections

Network reliability is a common problem faced in designing a system spread over multiple machines. Enterprises across the world spend a large amount of their time and resources on network management, but it remains that network links cannot be 100% reliable. Mission critical applications cannot afford to lose data in any eventuality and must always be built with this premise of unreliability. This requires that application programmers build a 'store and forward' layer in their Application infrastructures. This store and forward layer involves storing precious data upon detecting network loss, taking corrective action and resending the previously cached data again.

The JMS standard requires middleware to build the 'store and forward' mechanisms for consumers. This is achieved by marking a consumer 'durable'. If a consumer is unavailable, the server holds onto the messages. These messages are delivered when the consumer becomes available again. This standard JMS feature ensures that durable consumers always receive messages. JMS does not provide a similar level of reliability for a producer. If the server is unavailable the send mechanism of the producer fails, resulting in an appropriate exception. This forces applications to implement 'store' on the client side and transfer this data when connectivity is restored.

FioranoMQ enhances the capability of JMS support to provide the 'store and forward' function at the client end as well (in addition to the durable consumers on the server). This function allows JMS applications to continue all publish operations even if the server is un-available, freeing up applications from all network related problems. A network disruption is thus not visible for a JMS application built over FioranoMQ.

Overview

FioranoMQ introduces the concept of a 'Durable Connection'. A Durable Connection remains connected to the FioranoMQ Server at all time. Applications using durable connections do not have to store, re-connect and then forward stored messages to the server. This frees up the application from the complex task of building a 'store and forward' mechanism in application code.

The reliability of the underlying JMS transport is improved by durable connections. If the connection is lost and the application fails to transfer data, Durable Connections try to restore the connection automatically. This ensures that data is not lost in transit and is sent as soon as the connection is re-established. These activities are not visible to the application and are performed automatically by Fiorano's runtime library when it detects a connection failure. This makes the system reliable and robust even in the presence of network failures.

For example, consider a computer monitoring a steel mill. Real time steel production information is sent every second to a main hub. The main hub uses this information to generate the desired results. If the connection between the Process computer and the Hub breaks, the 'send' mechanism will fail and an exception is thrown. Since this data is generated only once, the application stores this data on encountering the exception and then applies its resources to connect back to the server. This process adds a considerable load to the application.

In such cases, a Durable Connection comes to the rescue as it does all the hard work on behalf of the application. It automatically tries to re-establish the connection, stores the data in transit and sends it to the server as soon as the connection is restored.

Note: Durable Connections are a proprietary feature of FioranoMQ. Durable Subscriptions are a part of JMS specifications.

Working of Durable Connection

A durable connection works like an ordinary connection as long as the connectivity is maintained with the FioranoMQ Server. If the underlying socket breaks, a durable connection performs the following activities:

- Initiates a thread that continuously tries to re-connect with the server.
- Initializes a 'store', on the local machine, to store (on the client side) any new messages published.

Both these activities are not visible to the client application and are performed automatically by FioranoMQ's runtime library. When a connection is restored, messages stored in the local store are automatically sent to the server.

Producer on a Durable Connection

A producer creating messages over a Durable Connection can send messages whether or not actively connected to the server. The runtime library automatically handles problems of connectivity. If the underlying connection breaks, the runtime library establishes a local cache of messages on the client's machine. This local cache stores messages published by the producer when disconnected from the server. The base directory of this local cache can be configured by the client application. A subdirectory for each connection using the cache is created in the base directory, where messages for particular connections are stored. The client application can use any number of Durable Connections over the same base directory. Once connectivity has been re-established via Fiorano's runtime library, messages stored in the local cache are transferred to the server. Messages are transferred in the same chronological order in which they were published.

An application is free to send messages to more than one JMS Destinations over a single Durable Connection. Producers can be created on transacted as well as non-transacted sessions.

Note: Messages are stored in the local cache, irrespective of their Delivery Mode. Persistent as well as Non-persistent messages are stored in the client side cache.

Consumer on a Durable Connection

Since consumers themselves can be defined as "durable" by virtue of their definition within JMS standards, very little is required to be done to ensure messages are delivered to a consumer, even if it that consumer is temporarily unavailable. If the consumer is created over a Durable Connection, the Fiorano runtime library automatically manages reconnects to the server in case of network failures. Message delivery is restored when the connection is established again.

Advantages

Durable Connections in FioranoMQ provide a host of advantages over standard JMS implementations:

Network Reliability

Durable connections provide network reliability by storing messages at the client end when the server is down – an essential feature required by most real-world systems.

Store and Forward Capabilities

Durable Connections enable 'store and forward' capabilities at client level.

Transparent Reconnection Code

If Durable Connections are enabled, the client application does not hold the responsibility for reconnection. Reconnection is handled internally by FioranoMQ's runtime library.

Message Browsing of Persisted Messages

FioranoMQ provides a Message Browser which allows messages stored in the data store of the client, to be browsed.

No Vendor Lock-in

Connection revalidation logic is transparent to the client application and is handled by Fiorano's runtime library. Reconnection code and other details do not have to be managed at the application level. Only the connectionfactory with AllowDurableConnection needs to be referred to. Client side persistence and reconnection code is handled in a transparent manner by Fiorano's runtime library.

Enabling Durable Connections Support

The ability to create a durable connection with the server can be controlled at the server and at the application level. Durable connections can be enabled/disabled using Fiorano Studio by:

1. Creating new connections to the server
2. Adding a new connection factory

Note: Disabling Durable Connection in server configuration disables it universally for all the clients. Disabling Durable Connection in a connection factory disables it for all the clients using the concerned connection factory.

Client side Message Cache

A Durable Connection creates a cache on the local machine, to allow a producer to send a message even if the server is unavailable. Configuration of the base directory is explained in the preceding section. Within the base directory, a subdirectory is created for each connection. The subdirectory's takes the name of the Client ID of the durable connection.

For example, if the base directory for Durable Connection is: `c:\\temp\\db` in `myConnectionFactory`, any connection created through myConnectionFactory creates its cache in `c:\\temp\\db`. If there are two connections on the same machine, with clientIDs "client1" and "client2", the directory structure takes the format below:

```
c:\\temp\\db

   |____ client1.ptp

   |____ client2.ptp
```

Note: If Client ID is not set, FioranoMQ at runtime internally creates a Unique ID for that particular connection and a directory by the same name is created for client side caching.

However, it is recommended that the Client ID should be set in all instances of use of Durable connections because: An application cannot transfer pending messages upon restart, as a new ID is generated for that connection. Since the ID is a complex string, it is difficult to use CSP Message Browser to browse for client side persisted messages.

Messages sent by a client are identified by its clientID. If the client application is terminated, , upon restart the runtime library checks if there are any pending messages stored in the local cache of the connection. This check is performed on the basis of the client ID set on the connection. Pending messages are sent to the server. This operation is performed when the clientID is set by the application. If the application wishes to ignore previously cached messages, it needs to add the following flag in the Hash table passed as the environment to InitialContext used for looking up operations.
"DONT_SEND_PREVIOUSLY_STORED_MESSAGES", "TRUE"

Use the following API available in the connection if the client application needs to exercise control over the time at which pending messages are to be transmitted:

```
public void sendPendingMessages ()
     throws JMSException;
```

When the preceding method is used on a connection, the runtime invocation sends all pending messages for that connection to the server.

```
public void purgePendingMessages ()
      throws JMSException;
```

The preceding method is used to purge all messages in the local cache published on the associated connection.

Note: Both the APIs require the casting of JMS Connection into fiorano.jms.runtime.ptp.FioranoQueueConnection or fiorano.jms.runtime.pubsub.FioranoTopicConnection.

Serverless Environment

It might be necessary in certain situations to run the client application in a serverless environment. A serverless instance is when a client needs to connect to a server even if the server is not available. This might be necessary when it is essential that a client connect with a server in situations where there is a high probability that a server is down Consider the case of a cellular service provider. The service provider has an SMS gateway that interacts with mobile devices by acting as an interface between the JMS server and the mobile phone. The JMS server routes the data ahead. A user uses his mobile phone/PDA or any other hand-held device to send an SMS to another user. This message first reaches the gateway, which has the responsibility of routing this message to another gateway interacting with the mobile device of the recipient, through a JMS server. There is a possibility that the server was down at the time the message is to be forwarded. If such a situation arises, the SMS would be lost.

Alternatively, if the gateway receiving the message from the sender is considered to be a JMS client with Durable Connections enabled, then it stores messages locally when the server is down. This provides a robust and reliable solution where messages are stored in the local cache and subsequently re-routed through the server when it becomes available.

To enable a client application to run in a serverless environment, the following needs to be set in an application:

```
env.put(FioranoJNDIContext.AllowDurableConnections, "true")
FioranoJNDIContext ic = new FioranoJNDIContext(env);
QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookupQCF("primaryQCF");
QueueConnection qc = qcf.createQueueConnection();
qc.setClientID("myClient");
QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWL-EDGE);
Queue queue = qs.createQueue("primaryQueue");
```

This allows client applications to run with Durable Connections enabled in serverless environments.

Note: Consider the case where the ‘lookup’ of the ConnectionFactory is performed using the `lookupQCF()` method and a queue is created using the `createQueue` method in the same session. When the server is running, the connection gets revalidated and if the ConnectionFactory exists on the server, the actual ‘lookup’ is performed through the server and messages are sent to the appropriate destination from the local cache. If it is found that the server does not allow durable connections after revalidation, then the pending messages are not sent to the server.

Sample Application

Sample applications are available in the following directories of the FioranoMQ installation:

```
%FMQ Home%\fmq\samples\ptp\Durable Connections
```

```
%FMQ Home%\fmq\samples\pubsub\Durable Connections
```

These samples can be downloaded from www.fiorano.com.

Relationship with Revalidate

In this model of Durable Connections, the client application does not hold the responsibility of reconnection. If Durable Connections is enabled, the entire process is handled internally by FioranoMQ's runtime library. If the client application has Durable Connections enabled, the client does not have to take care of the revalidation code if the server breaks down. The FioranoMQ runtime invocation detects network failure internally and starts a reconnection thread that reconnects client to the server when it's available.

Relationship with CSP

Client Side Persistence is provided in the 6.0 version and above. Proprietary APIs of FioranoMQ are needed for CSP.

To enable durable connections, client side persistence does not need to be enabled. In CSP, the following had to be set in the client application to enable client side persistence:

```
env.put(FioranoJNDIContext.ENABLE_CLIENT_SIDE_PERSISTENCE,"true")

env.put(FioranoJNDIContext.CSP_BASE_DIR,"c://myCache");

InitialContext ic = new InitialContext(env);

QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("primaryQCF");

QueueConnection qc = qcf.createQueueConnection();

(FioranoQueueConnection)qc.setCSPConnectionID("myClientID");
```

In Durable Connections these steps do not need to be performed. The client applications can 'lookup' a ConnectionFactory that has information on the Durable Connection. The application proceeds to set a unique clientID for the connection upon which Durable Connections are to be enabled. This enables messages sent by the sender to be stored in the directory structure specified in the ConnectionFactory (or in the ".\CSPCache" directory) and sent to the server once the connection is revalidated. Fiorano's runtime library handles the above steps, internally.

To enable durable connections, the client application is set to allow durable Connection in the env as a property, while setting a client ID for the connection:

```
env.put(FioranoJNDIContext.AllowDurableConnections, "true");

InitialContext ic = new InitialContext(env);

QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("primaryQCF");

QueueConnection qc = qcf.createQueueConnection();

qc.setClientID("myClientID");
```

Messages get stored on the client machine either in the directory specified in the env variable or in ".\CSPCache". Calling an explicit setCSPConnectionID on the connection on which CSP has to be enabled is not required. The only requirement entails setting a ClientID for the connection to enable Durable Connections.

Constraints in Durable Connections

Ensure that unique Client IDs are used if more than one application needs to use the same local cache simultaneously.

Using the browser for Client-side persistence while an application is using the same local cache can result in the browser behaving abnormally. The Client-side persistence browser should not be used while an application is using the same local cache.

Messages can get redelivered with the appropriate JMSREDELIVERED flag set. Messages can get redelivered when, for example, data reaches the MQServer and the client loses connection before the server acknowledges the receipt of data.

The base Durable Connection directory cannot be added while creating the connection factory through the Admin GUI.

Chapter 11: Hierarchical Topics

Developers need to organize their data based on its content. JMS accomplishes this by funneling messages to various destinations. These destinations can not have any logical correlation with each other.

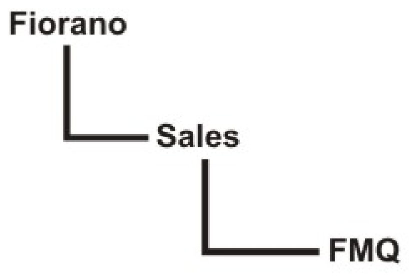
FioranoMQ provides a way of correlating various destinations. FioranoMQ destinations can have a parent child relationship. A topic can be created within a topic. This results in a hierarchical tree, where each leaf represents a unique topic.

Need For Hierarchical Name-Spaces

Topic name spaces offer the ability to organize various destinations in a hierarchical manner. An enterprise can choose to define various levels of hierarchy, depending on the organization of the data that needs to flow on these destinations. Hierarchical topics are easier for solution architects to visualize and design. Given there is a well-defined relationship within such hierarchies, it results in efficient handling of destinations by the provider as well. Even when a topic hierarchy is flat (linear), it is, typically, built from one or more root topics. This entails adding other topics in levels of parent-child relationships to create a hierarchical naming structure.

Name Space Notation

Hierarchical name-spaces of FioranoMQ use the same notation as qualified packages and classes. Various levels in the hierarchy are distinguished by period-delimited strings. For example, a topic name `fiorano.sales.fmq` results in the following hierarchy:



The hierarchy gets automatically defined at the time of creating the topic. The process of creating a topic is successful only when the parent topic exists. This ensures that nodes are added to the hierarchy tree in an orderly manner. FioranoMQ allows the hierarchy to have unlimited number of levels and unlimited number of nodes on a particular level.

Creating Hierarchical Topics

Hierarchical topics can be created like any other topic. A topic at a particular level can be created only if its parent exists in the hierarchy. The first node of a hierarchical topic is called the root node of the hierarchy. Some important features with respect to hierarchical topic names are noted below:

Case Insensitive

Topic names are not case sensitive in FioranoMQ. "ACCOUNTS" and "Accounts" are considered the same topic.

Spaces in Names

Topic names can include a ‘space’ as a character.. For example, "company.fiorano.comment.FioranoMQ is fast" is a valid topic name. Topic names are trimmed before creation. As such, even though spaces are allowed in topic-names, topics that differ only in the number of spaces they incorporate shall be considered to be duplicates. It is therefore recommended that spaces in topic names be used with appropriate caution.

Empty String

No level in the topic hierarchy can include an empty string. A topic name cannot have two simultaneous dots. For example, "`company.GE..dept`" is an invalid topic name.

Unlimited Length of Topic Names

Topic names can be arbitrary in length. A node in the topic hierarchy can have any number of characters.

Unlimited Depth of Topic Hierarchy

FioranoMQ supports an unlimited depth in the hierarchy tree of a topic. An unlimited number of nodes within a topic can be created.

Wild Card Support

Wildcard characters such as asterisk (*) or (#) cannot be used in topic names for creating hierarchical topics.

Dynamic Creation of Topics in Hierarchy

If a topic matching certain topics in a hierarchy is created on a running server instance and its name matches any subscription expression, then this topic becomes the member of the hierarchy.

Example: Subscription expression: ABC.*

Topics existing on system: ABC, ABC.1, ABC.2, ABC.1.1

A subscriber looks up topics with expression ABC.* and it is receives messages from all matched topics. If at runtime a new topic named ABC.3 is created, thenABC.3 becomes a part of the hierarchy. Published messages on ABC.3 are received by the Subscriber created on ABC.*.

Note: For this feature to function, events must be enabled at the server end.

Looking up Hierarchical Topics

Client applications can lookup a topic in the FioranoMQ Server using either JNDI APIs or a bound object of type FioranoInitialContext.

Criteria for looking up hierarchical topics are:

The topic being looked up contains a wild-card character, either (*) or (#), along with any number of delimiters (.). The lookup call succeeds only if the root topic is created by the administrator earlier. If the topic being looked up contains a (*) or (#), the call is successful only when there is at least one topic in the server whose name matches the named looked up.

For example: If the user tries to lookup "primarytopic.a.*" or "primarytopic.a.#", then the lookup is successful only if "primarytopic.a" exists on the server.

Publishing on Node(s) in Topic Hierarchy

A Publisher can publish only on fully specified topic names. Publishing on a topic that contains an asterisk (*) or a pound character (#) throws an exception.

Subscribing to Node(s) in Topic Hierarchy

Subscriptions are created as defined by JMS using TopicSessions. The normal `createSubscriber` APIs, provided by JMS, can be used to create subscriptions on hierarchical topics.

A subscriber can subscribe to multiple topics using a wild-card character. Subscribing to a topic containing a valid wild-card character effectively creates subscribers on all the topics in the hierarchy that match that expression.

Template Characters Used in Subscription

Wild-card characters are special characters used in the creation of topic hierarchies. In the topics hierarchy, these characters are referred to as Template Characters. The period (.) delimiter is used together with the asterisk (*) and the pound (#) template characters to fulfill subscriptions. Using these characters avoids having to subscribe to multiple topics on the server. Client applications can use template characters when subscribing to a set of topics or while binding a set of topics.

There are two FioranoMQ template characters used in subscription-creation; the asterisk (*) and the pound (#):

- **Asterisk (*)**: FioranoMQ uses two types of conventions for this template character An asterisk (*) must be the last template character in subscription expression. Subscriptions are made for the root node and all matching subordinate nodes in the hierarchy.

  For Example: If the expression for subscription is ABC.*, then ABC and all its subordinate topics will be matched.

  An asterisk (*) is the intermediate character within a subscription expression. Where a root topic is not selected an (*) is taken as "one or more occurrence of a character ". Example: If the topic for subscription is ABC.*.1 then ABC will not be selected but all topics that match this pattern will be selected and used to subscription. Example: ABC.1.1, ABC.1.1.1, and ABC.2.1.
- **Pound (#)**: The pound (#) selects all topics one level down the hierarchy. If the (#) character is present in the pattern, then all the topics one level down the hierarchy are used for subscription. Example: If the subscription topic name is ABC.# then all the topics a level below ABC will be matched. Example: ABC.1, ABC.2 will match ABC.#, but ABC.1.1 and ABC.1.2 do not match the pattern and will not be used.
- Template characters exist to allow a set of managed topics to exist in a message server. This, in turn, allows the subscriber to choose broad subscription parameters that include preferred topics and avoid irrelevant topics.

The constraints in using template characters are

At node level, a template character precludes using other template characters. Example: Qualifying the selection against the pattern A.B*.1 is not allowed where as A.B.*.1 is allowed. In the patterns used, each character must be separated with a delimiter (.).

Template characters used as replacement are not allowed.

Other than the wild-card characters (*), (#), (.), no other wild-card character is used for subscription to multiple topics.

Conventions used in Hierarchical topics: Only two template characters are used in Fiorano hierarchical topics. These characters are the pound (#) and the asterisk (*) with a delimiter (.).

Using the asterisk (*)

Subscription Expression ABC.*

Convention used: If (*) is the last character in a subscription expression with no other template character used, then the root topic and all other topics (where (*) is replaced with one or more occurrences of any character) will be selected for subscription on Hierarchical topics.

Example:

If the following Topics exist on the MQ Server: ABC, ABC.1, ABC.2, ABC.1.1, and ABC.1.2

And the expression used for subscription is:

```
ABC.*
```

Then the matched topics are:

```
ABC, ABC.1, ABC.2, ABC.1.1, ABC.1.2
```

Subscription topic name format:

```
ABC.*.1
```

Convention used: If (*) is the intermediate character in a subscription expression then all other topics (where * is replaced with one or more occurrences of any character in name) will be selected for that subscription. The root topic is not included in such a selection.

Example:

The the Topics existing in MQ Server are:

```
ABC, ABC.1, ABC.2, ABC.1.1, and ABC.1.1.1
```

And the expression used for subscription is:

```
ABC.*.1
```

Then the Matched topics are:

```
ABC.1.1, ABC.1.1.1
```

Using the pound (#) character

Subscription topic name format:

```
ABC. #
```

Convention used: If (#) is the only wild-character present in the expression then all topics (where (#) is replaced with only one occurrence of any character in name) would be used for subscription.

Example

If the Topics existing on the MQ Server are:

```
ABC, ABC.1, ABC.2, ABC.1.1, ABC.1.1.1
```

And the expression used for subscription is:

```
ABC. #
```

Then the Matched topics are:

```
ABC.1, ABC.2
```

If the Subscription topic name is ABC#.1,

Then the Matched topics are:

```
ABC.1.1, ABC.2.1, ABC.3.1
```

Using a combination of template characters: Both template characters can be used in a subscription expression, as explained below.

Subscription topic name format:

```
ABC.*.#
```

Convention used: The above topic name is invalid. The pound (#) character after the asterisk (*) character has no function in an expression.

Subscription topic name format:

```
ABC.#.*
```

Convention used: In this expression all the topics (where (#) is replaced with one occurrence of a character and (*) is replaced with one or more occurrence of a character) will be used for subscription.

Example:

If the Topics existing on the MQ Server are:

```
ABC, ABC.1, ABC.1.1, ABC.2, ABC.2.1
```

And the subscription expression is:

```
ABC.#.*,
```

Then the Matched topics are:

```
ABC.1.1, ABC.2.1
```

Deleting a Hierarchical Topic

Deletion of a topic/subtopic from the hierarchical 'name space' depends on the value of the parameter `AllowDeletionOfSubTopics`, which can be configured through Fiorano Studio. If this value is set to true, then deletion of a topic/subtopic deletes all the children of this topic/subtopic. If it is set to false, an exception is thrown indicating that the user first needs to delete the children of the topic\subtopic before deleting the topic itself. By default, this variable is set to false.

Publish/Subscribe Across Servers

FioranoMQ supports hierarchical topics across servers. Hierarchical topics across servers can be used in exactly the same way as they are used on a single server.

Security Considerations on Hierarchical Topics

FioranoMQ supports ACL settings for hierarchical topics. An ACL can be set for any topic, irrespective of the level at which this topic exists. These ACLs are checked at the time of creation of a publisher and/or subscriber. When creating a subscriber for multiple topics (a topic that involves a template character in its name), the ACLs for all the subtopics are also checked. In addition, the subscriber is modified so that it does not receive messages from all the subtopics that have a negative permission set for that particular user.

Limitations

Topic names cannot contain a wild-card character in topic creation. For subscription expression on the template characters ((*) or (#) with any number of delimiters (.)) can be used. Usage of any other template character throws an exception in 'lookup'.

Deletion of a hierarchical topic that has active publishers/subscribers is disallowed. Care needs to be taken not to delete the hierarchical topics when any topic in the hierarchy contains active publishers or subscribers.

A publisher cannot publish on multiple hierarchical topics concurrently. A publisher has to specify the complete name of the hierarchical topic on which it wants to publish data. Creation of a publisher on a topic that contains an asterisk (*) throws an exception. Similarly, an exception is thrown if a publisher tries to publish on a topic which contains an asterisk (*).

If a subscriber subscribes on hierarchical topics with a subscription expression, and the ACL of a 'child' in the hierarchy is changed by the administrator mid way through receiving messages, the subscriber remains unaffected by this change. However, creating new subscribers with subscription expressions will get affected by this change.

There is a performance degradation associated with hierarchical topics. Users are advised not to use hierarchical topics for applications where performance is a major requirement.

Chapter 12: Message Encryption

Message encryption allows transfer of sensitive data from one point to another in a secure way. Encryption implies the transformation of plain text into cipher text which is not possible to read without the use of a "key". A key is also used to decrypt the cipher text into plain text.

Base Implementation

FioranoMQ versions 7.1 and upward support encryption. DES (Data Encryption Standards) is used as the default encryption algorithm. FioranoMQ intends to support more encryption algorithms in its future releases.

There are two types of encryption algorithms:

- secret key algorithms
- public key algorithms

In secret key algorithms, both the sender and the receiver need the same key for encryption and decryption. In public key algorithms, the public key is used for encryption that is published and the private key is used for decryption so that No secret information is exchanged. The private key is mathematically related to the public key. Theoretically it is possible to compute the private key based on the public key. To avoid easy computation of the private key by unauthorized third parties, the computation is made as complex as possible. DES is based on a secret key cryptography.

An advantage of the secret key cryptography compared to public key cryptography is the faster speed of computation. Therefore, this method is recommended for bulk encryption and is commonly used over other methods. The encrypted text is compact.

The disadvantage of secret key cryptography is that the administration of keys can become complicated because key sharing.

In setups where imparting key information happens in a secure way, secret key cryptography can be used. Public key cryptography is supposed to make secret key cryptography more secure and is used when such a need exists.

The message encryption function uses the library `cryptix.jar` provided by Cryptix for generating keys as well as for encryption. This file comes bundled with the FioranoMQ installation. It can be found in the `FIORANO_HOME%/extlib/cryptix` directory of the FioranoMQ installation.

Message Encryption Characteristics

FioranoMQ provides message encryption on ‘per message’ as well as on ‘per destination’ basis.

In ‘per message’ encryption, clients can enable or disable encryption for each message. ‘Per message’ encryption is done by a client before relaying data to the network. Decryption must be performed by the receiving client application prior to reading the message.

In ‘per destination’ encryption, all messages sent to a particular destination (topic or queue) are encrypted, thus providing a secure channel of delivery. A destination is marked as encrypted at the time of its creation. All messages published on this destination is delivered decrypted to subscribing applications. A client application, therefore, does not have to explicitly decrypt a received message.

Encryption involves only encrypting the payload of the message and not its JMS header. This allows usage of the same set of APIs associated with message headers as well as message selectors, irrespective of whether message encryption is enabled.

Chapter 13: Message Compression

Message compression is a function that allows messages sent through FioranoMQ to be compressed when sending and decompressed, to their original size, prior to delivery to consumers.

Compression has the advantage of improving performance. Less bandwidth is used during message transfer. Memory and storage requirements on the server are reduced as well. This function is important for performance-sensitive applications operating over WAN links. Fiorano also extends compression support for server-to-server communication.

Base Implementation

Many data compression implementations have been developed in the past, of which the Zlib implementation is, by far, the most significant one. The Fiorano compression implementation is based on "Zlib Compressed Data Format Specification Version.

This specification defines a lossless compression data format. The advantages of this compression implementation, as per specification, are:

- It is independent of CPU type, operating system, file system and character set.
- Can be produced or consumed by an arbitrarily long sequentially presented input data stream, using a bounded amount of intermediate storage.
- Can be implemented readily in a manner not covered by patents.
- Can use a number of different compression methods.

In FioranoMQ, the Zlib implementation provided in the default Java runtime library `java.util.zip` has been used. This implementation provides 'deflate' and 'inflate' mechanisms using different compression levels and different compression strategies. Compression level is the amount of compression required. Compression strategy is the actual compression method used.) The default strategy uses a combination of the LZ77 algorithm and Huffman coding.

Message Compression Characteristics

FioranoMQ provides message compression on a 'per message' as well as on 'per destination' basis. In 'per message' compression, clients can enable or disable compression for each message. In 'per destination' compression, all messages sent to a particular destination (topic or queue) are compressed.

Client applications can choose compression levels and strategies from Zlib specifications using public APIs.

The available options are:

- NO_COMPRESSION
- BEST_SPEED (fastest compression)
- BEST_COMPRESSION
- DEFAULT_COMPRESSION

There are ten possible compression levels (0-9) available, where BEST_SPEED is defined as 1 and BEST_COMPRESSION is defined as 9.

The possible values for the compression strategy are:

- FILTERED: Compression strategy best used for data consisting primarily of small values with random distribution. It enforces more Huffman coding and less string matching.
- HUFFMAN_ONLY:
- DEFAULT_STRATEGY: This uses a combination of the LZ77 method and Huffman coding.

Compression support provided helps a client application to decide on the optimum compression level and strategy by providing APIs to check compression ratios of messages sent and/or received.

Compression involves compressing only the payload of the message and not its JMS header. The same set of APIs can be used for message headers as well as message selectors, irrespective of whether message compression is enabled or not.

FioranoMQ's implementation allows users to plug in their proprietary compression implementation, which overrides the default implementation.

Chapter 14: FioranoMQ Clustering

In real-world applications, it is a common to manage a heavy load of connections over messaging servers. Installing the messaging server using the best hardware available alone will not suffice. There must be a more scalable approach to handling a linearly increasing number of connections.

The most appropriate solution is to have 'n' number of messaging servers communicate with each other, share the load between them and work in synchronization. A logical unit consisting of these 'n' servers (and some other software components) is called a cluster. A cluster also provides support for failover, which is not feasible with a single server.

Common Problems of Real-World Systems

This section gives an introduction to problems faced in distributed systems in real-world scenarios.

Client Unable to Connect

In certain situations, a server can be temporarily unable to accept a connection request. An example could be socket buffering that occurs if too many clients try to connect at the same time. The most desirable behavior for the client is to transparently attempt a few reconnections first.

Since the server can be completely down rather than temporarily overloaded, the client needs to be able to connect to alternate backup servers. If this list of backup servers is retrieved as a parameter of a `connectionfactory` object, the client code can become non-portable.

Connection if the Server is Lost

This again is an important failure that should be handled. Client side persistence is a requirement. This ‘store and forward’ feature enables a client to operate in a disconnected mode, avoiding loss of messages. A seamless integration of client-side persistence should be transparent, allow for transacted sessions and cater for duplicated messages.

The Server Runs Out of Resources

There are many resources that can effectively render a server inaccessible because of their shortage: connections, RAM, disk space, threads, file descriptors, sockets, and possibly, others. A cluster of servers can provide more resources, distribute requests more evenly (load-balancing) and configure servers as ‘standby’ and ‘ready to take over’ in case of an emergency. To preserve application portability, the cluster should appear as a single (super) server where: load balancing and failover are transparent to clients. At times, the shortage of a resource can be temporary and it can be advisable for a client to first try and reconnect for a while before looking for an alternate server. Again, such an option should be compatible with load balancing.

The Server Goes Down Altogether

If a server crashes, clients connected to it need to be able to continue working by connecting to a secondary server. This scenario is termed ‘failover’. Once a server recovers, it needs to be reactivated for taking over the tasks assigned to it thereby restoring it to the state before the crash. It is termed ‘hot failover’ if processing can continue seamlessly (with nearly no latency). This requires that a secondary server is running and has access to persistent state and message data.

FioranoMQ: The Solution

FioranoMQ provides the following features to solve the above problems:

Automatic Failover Protection

Failure of a component should not cause failure of the whole system. The backup of important components is required to provide high availability. There should be no single point of failure. FioranoMQ's runtime invocation ensures that clients connected to a server are automatically ‘failed over’ to the back-up server. When an application wants to connect to a particular server, N attempts are made to connect to that server. If a connection still fails, the runtime library tries the URL of backup servers specified in the connection factory used to open the connection. Since this operation is performed automatically by Fiorano's runtime library, the application does not fetch the list of backup URLs. This avoids any vendor lock-in as the application code is based on pure JMS.

Transparency and Code Portability

The automatic failover protection mechanism should be completely transparent to the client. FioranoMQ provides this transparency by making the reconnection mechanism invisible to the client. The reconnection is done automatically by the runtime library. This is achieved via server configuration and does not require the use of any proprietary APIs making the client code completely portable across JMS server implementations.

Configurability

FioranoMQ achieves a high degree of configurability through its modular design. Interfaces for a number of modules are available to the public, which allow a developer to implement their own version of the module and plug it in. For example, a developer is free to write a Log Module that displays information in a Java Frame instead of the java console and plug it in. Besides these modules, a number of configurable flags are provided to the developer through the server's config file, which allows a developer to tweak various parameters of the server. For example, FioranoMQ provides a configurable option for the maximum number of clients waiting to connect to the server. The default number is 500. This figure can be increased or decreased through the server's configuration file. Note that the numbers specified represents the length of the queue of pending sockets and has no relationship with the maximum number of simultaneous clients that FioranoMQ can support.

Admin System

Availability of the system holding the administered objects (admin system) must be provided in the first place. The admin system should be different from the client systems and it should be backed up by a secondary source. Client access to primary or backup admin system should be transparent. FioranoMQ allows the administrator to configure the admin system storage as and when required.

The administrator can choose the storage media of administered objects from the following options:

- JNDI compliant Directory Server
- RDBMS Server
- XML File
- Fiorano's proprietary file format (that also uses JNDI)

Despite these options, a developer is free to write his/her own customized version of admin system. Storing the admin objects in a central Directory Server allows a client to directly lookup these objects through JNDI. The client need not go through the FioranoMQ server (though it is possible to do so). This makes the admin system independent of the server.

Connection if the Server is lost

FioranoMQ provides a client with the ability to automatically connect to a failover or backup FioranoMQ server. This mechanism works if the client's existing connection is broken or if the client's primary server is unavailable at the time of creating a connection.

This mechanism is transparent to the client application. In case an existing connection breaks and there is no backup URL specified, the client application can continue its routine operations of pushing more messages. These messages are stored in a local repository on the client machine. Meanwhile, the runtime library automatically tries to re-connect back to the server in the background, periodically. Once the connection has been established, the runtime system automatically transfers all the messages stored in the local repository to the server. This provides the client application with the 'store and forward' facility for a publisher in those situations when the connection to the server is temporarily broken. More importantly, this does not require the use of any proprietary APIs and hence avoids vendor lock-in of any kind.

Server Runs Out of Resources

FioranoMQ's dispatcher is the load-balancing component within the Fiorano Server. The dispatcher can be enabled or disabled easily through the server's configuration file. If enabled, the server distributes incoming connections to members of its clusters. Again, this does not require special APIs for the client application. The application only sees the dispatcher-enabled server. The dispatcher administrator is free to add/remove members in the dispatcher cluster any time. If a member server in the cluster goes down for to any reason, all the applications connected to this server shift to one of the other members in the cluster. FioranoMQ also provides fail-over for the dispatcher server, which means that users can setup a secondary dispatcher that is used in cases where the dispatcher server goes down. The client system can have the URL of the secondary dispatcher as its backup URL. When the primary dispatcher goes down the client system gets automatically load-balanced by the secondary dispatcher.

An important requirement for running FioranoMQ Servers in a cluster is that TopicConnectionFactories (TCF), Topics, QueueConnectionFactories (QCF), Queues and UnifiedConnectionFactories (CF) should exist on all servers running in the clustered environment and on the server acting as a dispatcher. These components are used by the clients to make connections to the least loaded FioranoMQ server through the dispatcher. The TCFs, QCFs and CFs replication amongst servers is required to make the new connections to the server. The Topic and Queue replication is required for automatic failover support for clients. If the client connection to any of the servers in the cluster goes down, the client gets connected to another server running on the cluster.

Server's Connection if a Client is Lost

If a connection with a consumer gets broken and if the consumer is durable, the server continues to store messages for the consumer published by the producer. These messages are made available to the consumer next time it logs into the system. Fiorano's runtime library also pings the connections to the server periodically (with a configurable time difference between two pings), which allows the server to detect dead sockets and clean them up. This becomes a lifesaver in case of network failure, which is not detected by the JVM unless a write operation is performed on it, which is usually not the case.

Server-to-Server Communication

It is a common requirement of real world applications to allow clients to exchange information seamlessly across servers. The Repeater and Bridge components of FioranoMQ are used for server-to-server communication over topics and queues respectively. Apart from FioranoMQ to FioranoMQ server communication, bridges are available for other messaging servers including: IBM WebsphereMQ, MSMQ, and Tibco Rendezvous.

Scalability

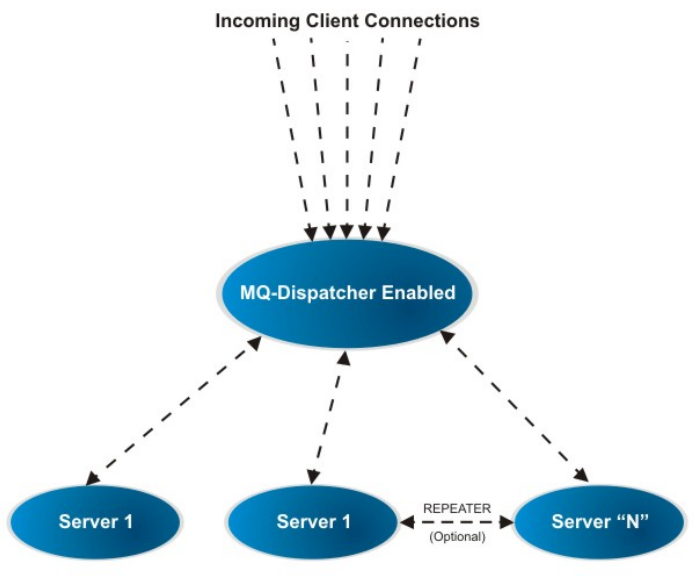
The load balancing and failover protection architecture of FioranoMQ Server allows unlimited scalability in terms of the number of client applications that can concurrently access JMS services. Thousands of concurrent client connections can be supported by a single cluster of servers. Combined with server-to-server communication, the Fiorano clustering architecture provides a very robust solution for a vast set of customer problems. For handling thousands of concurrent client connections, FioranoMQ also provides scalable connection management.

Clustering Components

FioranoMQ provides extensive clustering support using components such as dispatcher, repeater, and bridge. The dispatcher is used for load balancing (distributing load) client connections among different servers running in a clustered environment. The repeater and bridge are used for server-to-server communication over topics and queues, respectively. Apart from FioranoMQ to FioranoMQ server communication, bridges are available for other messaging servers including: IBM WebsphereMQ, MSMQ and Tibco Rendezvous. A detailed explanation of these components is provided in subsequent sections of this chapter.

Dispatcher

FioranoMQ's load balancing architecture involves the use of a Dispatcher-enabled MQServer to route incoming client connections to the least-loaded server in a cluster, as illustrated in the figure below.



The dispatcher component of a FioranoMQ server is typically connected to multiple FioranoMQ servers, which can run on different machines or on the same machine. All of these servers become part of the "cluster" that is serviced by the dispatcher.

The FioranoMQ Dispatcher runs as part of a server process and maintains a persistent connection with each FioranoMQ Server in its cluster. This persistent connection is used to pass information from the server to the dispatcher, enabling the dispatcher to maintain real-time "in-memory statistics" about the precise load in terms of the number of connections on each server. The dispatcher uses this information to determine the least loaded server in the cluster and route new incoming client requests accordingly.

An advantage of using the Fiorano Dispatcher is that no changes are required on the Client application in order to use Dispatcher. Once this function is turned on in a server, it automatically routes connection requests to the least loaded server. The server load is calculated internally by the dispatcher based on the maximum connections allowed on a particular server, and the number of active connections.

Preferred Server

At times, a particular client application may want to connect to a particular server in a cluster. This can be done by setting a flag to the "preferred server" within the cluster in the connection factory being used or in the lookup environment. The preferred server can be set through dispatcher configuration. The preferred server is typically used by client applications that have created durable subscriptions on a particular FioranoMQ server within known server clusters and wish to reconnect to the same server to retrieve messages.

Configuration Parameters

Parameter	**Description**
Login Name	Represents the login name used by the dispatcher to connect to a MQ server that parties a member of a cluster. . The login should have admin privileges.
Password	Represents the password used by the dispatcher to connect to a member MQ server.
AdminConnectionFactory	Specifies the admin connection factory used by the dispatcher to connect to a member MQ server.
Server URL	Specifies the URL of the server in the cluster (Format: http://hostname:port) Server Admin URL specifies the admin URL of the server in the cluster (Format: http://hostname/port)
Backup Connect URL	URLs of the backup servers used in case the primary server is down. Multiple backup URLs can be specified as a string of URLs separated by a semicolon. Example of a Backup Connect URL is http://backupServer1:1856; http://backupServer2:1856
MaxClientConnections	Specifies the weight associated with a server that is part of a cluster. A server with MaxClientConnections set to 2 will allow twice the number of connections as that set to 1.
SecurityManager	The Security Manager implementation is used to create secure connections with the MQ server. The manager should be an implementation of the `fiorano.jms.runtime` interface provided by FioranoMQ.
TransportProtocol (TCP/HTTP)	Is the protocol used for communicating with a server. Transport protocol can be set to either TCP or HTTP.
java.naming.security.protocol	Name of the security protocol used to create secure connections with the MQ server. The possible values that this variable can take are PHAOS_SSL and SUN_SSL.

Table: Configuration Parameters

Repeater

FioranoMQ architecture allows multiple FioranoMQ Servers to be connected together, allowing clients connected on one server to exchange information with clients connected on any other MQ Server. Using FioranoMQ Repeater, servers can be connected both over LAN (Local Area Networks) and WAN (Wide Area Networks).

FioranoMQ Server clustering allows clients connected on different FioranoMQ servers to exchange information by setting up an instance of the FioranoMQ Repeater. This feature is particularly useful in deploying applications that need to communicate with other applications across geographically distributed sites. For instance, take the example of an organization with offices in New York, San Francisco and Boston. Here, a client of a MQ Server located in the Boston office can communicate with a client of a MQ Server located in the New York office. This is easily achievable with server-to-server communication, facilitated by a repeater. With Server-to-Server communication, each client application in Boston only needs to connect to the Boston FioranoMQ Server.

The FioranoMQ Administrator can configure the Repeater to automatically forward relevant messages from the Boston server to the FioranoMQ Server in New York and/or San Francisco, based on specified requirements. These messages are delivered to the subscriber applications that are connected to the New York and/or San Francisco servers. If there are transient network failures across the WAN connecting Boston, New York and San Francisco, none of the client applications are affected. Publishers can continue to publish messages locally. These messages are persistent on the local server of the publisher if a durable link (For more information, read the Subscription Mode and Choice of Selectors section) on the repeater connects the concerned FioranoMQ servers. The subscribers stay connected and receive messages, if any are available, from their local server. The repeater also takes care of reconnecting the servers in case of temporary network failures.

FioranoMQ Repeater enables the communication between different servers by using the Publish/Subscribe messaging model. This implies that information is exchanged between the topics on different servers and the repeater cannot be used for exchange of information over queues. All Server-to-Server communication is handled in a transparent manner by FioranoMQ internally and the client application does not need to be modified in any way. MQ Servers can be part of the same LAN or can be spread across multiple WANs.

The FioranoMQ Repeater allows information exchange over SSL/HTTP/HTTPS in addition to the default TCP/IP communication.

Salient Features

FioranoMQ Repeater offers the following features:

- **Easy Configuration**: FioranoMQ Repeater can run as embedded in the same container in which the server is running, or can run as a standalone component (separate process) and can be used to wire multiple servers. FioranoMQ Repeater provides complete power to the enterprise administrator to configure MQ Servers on any network topology. An administrator can set up topics on source as well as target servers that exchange information between them. Configuring the Repeater is simple and is XML based.
- **Connection Topology**: The Administrator of FioranoMQ can configure the Repeater to set up a connection between servers and propagate messages on the connection.

The repeater can be configured to support different kinds of network topologies:

- **Hub-spoke:** A source server can be linked with N number of target servers and vice-versa. In the former case, the single source server acts like a message broadcast hub for all target servers. All messages published on the source server can reach one or all of the target servers depending on the requirement. Similarly, in an opposite scenario, a single target server can act as a hub for many source servers and receive messages published on one or all source servers.
- **Mesh**: A cluster of FioranoMQ servers can be set up in a manner where each server is connected to all other servers in the cluster through the repeater, forming a mesh type of structure. Messages published one server can reach other servers in the cluster, depending upon the topics on which the messages are published.
- **Bus based**: The repeater can be used to set up a cluster of FioranoMQ Servers, in which a single source server represents a message bus. Target servers behave as recipients of messages from this bus where only one message can be received by only one target server at any point of time.

No Changes in the Client Application

Client applications can communicate with any number of FioranoMQ Servers connected through the Repeater. Clients connected to one server can exchange messages with clients connected on another server, without each client having to explicitly connect to the same server. The client application does not need to be modified in any manner - the FioranoMQ Repeater forwards messages pertaining to a particular topic across servers. The administrator only has to configure the repeater to forward messages pertaining to desired topics across servers.

Robustness in Handling Network Failures

Data transfers between multiple FioranoMQ Servers (connected to each other through the Repeater) can be made to use Persistent Messages/Durable Subscriptions as an option. In this case, messages transferred between servers are always logged onto persistent storage, thereby making the system highly reliable and robust in the event of network failure.

In the event of a network connection going down, the repeater tries to bring it up. The repeater tries to reconnect to the server with which it lost a connection repeatedly, with only a small interval between each try. This ‘ping’ time interval is configurable through Fiorano Studio. The pinging operation continues until a connection is re-established (where the "down" server finally comes up again).

Subscription Mode and Choice of Selectors

The FioranoMQ Administrator can configure the Repeater to create either Durable or Non-Durable links between the source and target servers. A durable link can be used to ensure that no messages are lost across the repeater in case of network failure.

Message selectors can be set on a link between servers to allow only the required messages to be exchanged between them. These can be useful, especially in setting up the bus-based network topology. For more information, read the Connection Topology section.

Request/Reply Across Repeater

The Repeater provides functionality for using request-reply service over FioranoMQ servers, which are linked by repeater. This allows a requestor, publishing request messages on topic t1 on server S1 to get replies for the requests. These replies are received from a replier on topic t2 on server s2. The repeater can be set up, with little effort, to perform a request-reply scenario.

Dynamic Replication Links

FioranoMQ can be used to create new replication links dynamically. This enables the applications to replicate messages on topics that are created after the repeater has started. Administrators do not have to manually add replication links for all topics. They need only specify a pattern - 'ABC*' - and the FioranoMQ repeater would create replication links for all topics that matches the pattern. If new matching topic is created after the repeater starts, a replication link for that topic is created dynamically.

Repeater with Load Balancing

The repeater can be put to best use in a Load Balanced cluster of FioranoMQ Servers. FioranoMQ uses the dispatcher for load balancing client connections among different FioranoMQ Servers.

Repeater Link

The repeater replicates messages in the link specified between a source and a target server. A repeater can have N number of links configured. By default, the server sets up only a single link to the repeater. Properties related to this default link can be edited prior to creating and managing additional links in the online mode. The Link element, within the Repeater Manager MBean provides, has the following information:

- **Status:** Specifies the link is running or not.

  Note: This is not a read-only parameter and its value can't be edited through any tool.

- **SourceServer**: Specifies the server on which subscriptions are created. Source Server contains the ConnectionInfo.

- **TargetServer:** Specifies the server on which publishers are created. Target Server contains the ConnectionInfo.

Connection Information

The Connection Information enables the repeater to connect to source or target servers. Target Server as well as Source Server within a link is associated with an instance of Connection Information. An instance of ConnectionInfo contains the elements as listed in the table Configuration Parameters.

Link Topic Information

Each Link could be associated with one or more Topic Links. Each Topic Link refers to source/target topic information. A repeater picks up messages from the source topic and sends them to the target topic.

A link could also have an instance of a Request/Reply Topic. Information regarding the source and target topics are request-reply services only. They are used when LinkTopic uses a request-reply message transfer.

Configurable parameters of `LinkTopicInfo` are summarized below:

Configuration Parameters

Topic Link Information Parameters	
Source Topic Name	Specifies the name of the topic on which subscriptions are made on source server of the link containing `LinkTopicInfo`. The name supports the wild-card character '*', which enables the repeater to create subscriptions on all the topics that matches the source topic. For example, if the source topic name is 'ABC*', subscriptions will be made on topics such as ABC1, ABC12, ABCDEF, and so on.
Target Topic Name	The name of the topic on which messages received for the above subscription are forwarded to the target server of the link containing `LinkTopicInfo`
ReplyOn	Specifies the topic name on which the repeater listens for replies which it receives on requests it forwards on `LinkTopicInfo`.
isDurable	Specifies whether the link between the source and the target is durable. A durable link ensures that no messages are lost across the repeater in the event of a network failure. The values for this variable are "yes' and "no".
Message Selector	Specifies the selector that is set on a link between servers to allow only required messages to be exchanged between them.
Type	Specifies whether the link should be constantly connected to the target server or replicated only if a subscriber exists.

ReplyTopicInfo Parameters	
ReplyTopicName	Specifies the name of the `ReplyTopic`
isDurable	Specifies whether the link between source and target is durable. A durable link can be used to ensure that no messages are lost across the repeater in the event of a network failure. The values for this variable are “yes’ and “no”.
Message Selector	Specifies the selector that is set on a link between servers to allow only required messages to be exchanged between them.

Wild Character Support

FioranoMQ provides support for wild-card characters in repeater configurations so that separate links need not be added for each topic. A user can specify wild-card characters in the source topic. All topics starting with the string mentioned in the source topic can be repeated.

A Repeater can be configured to replicate messages that match a particular pattern. This pattern can be specified in the source topic name of the Properties Name. For example, if the Source Topic Name is specified "ABC*", the topics that match this pattern (all the topics starting with the string "ABC" on the source server) are repeated across two servers. All subscribers subscribing to ABC, ABC1, ABCZ and so on will be able to receive messages published on source topics ABC, ABC1 and ABCZ respectively, via the FioranoMQ repeater.

Topics created dynamically that matches the pattern, 'ABC*' are replicated as well. Where 'ABC2' gets created after the repeater starts leads to creating, dynamically, a replication link for 'ABC2' (topic on source server) to 'ABC2'(topic on target server). And where a topic name (like 'ABD1') that does not match the pattern ('ABC*') gets created, the replication link is not added.

Dynamic Link Propagation

The Repeater can be configured to replicate messages only on demand, that is, messages would travel from source to target only if there is a consumer (active or passive) on the target server interested in the message. A pre-configured link in the repeater remains in a "stopped" mode if there are no consumers on the destination. This link is activated as soon as a consumer is created. By default this feature is turned “off”.

Note: For this function events need to be turned on in the server connected to repeaters.

Request / Reply across Servers

The FioranoMQ Repeater provides a mechanism for using request-reply service across two servers. An intermediate topic on the target server is needed so as to receive replies from the replier and forward them to the requestor.

A Possible Scenario:

- A requestor (REQ) connected to FMQServer 1, is publishing request messages on topic t1. The requestor awaits a reply for requests on a temporary topic created for the connection.
- A replier (REP) connected to FMQServer2, is subscribing to request messages on topic t2. On receipt of messages, the replier sends a reply to the request message on the reply topic that is specified in the `JMSReplyTo` property of the request message.
- Using repeater, the requestor REQ on topic T1 of Server1 can get replies for requests made by it.
- Specify the replytopicname as T3 using Fiorano Studio.
- Create a topic T3 on the Server2 (target server)

Refresh the Repeater

The SourceTopicName and the TargetTopicName can have the same name. Similarly, the replyOn value signifies the topic that is used by the target server for publishing replies to messages. This is for messages forwarded on that topic link, represented by LinkTopicInfo. The ReplyTopicInfo represents information about the replier topic link used in request-reply services across servers.

In addition, a permanent topic t3 has to be established on the target server if the reply is to be sent to a temporary topic on the source server. This is because the repeater requires a topic on which it listens to replies for requests that it has forwarded.

Bridge

Due to lack of a standard communication protocol, every messaging vendor invariably uses a proprietary protocol for client-server data-exchange. Lack of "interoperability" across multiple messaging vendors poses serious problems for communication across messaging systems. For instance, FioranoMQ applications cannot push or pop messages to IBM WebSphereMQ queues and vice versa. This lack of interoperability among message queuing systems can be a problem for businesses that merge, particularly when they want to integrate information systems based on different message queuing systems. FioranoMQ solves this problem by "bridging" FioranoMQ with IBM MQ Series, MSMQ, Tibrv and all other JMS providers' products.

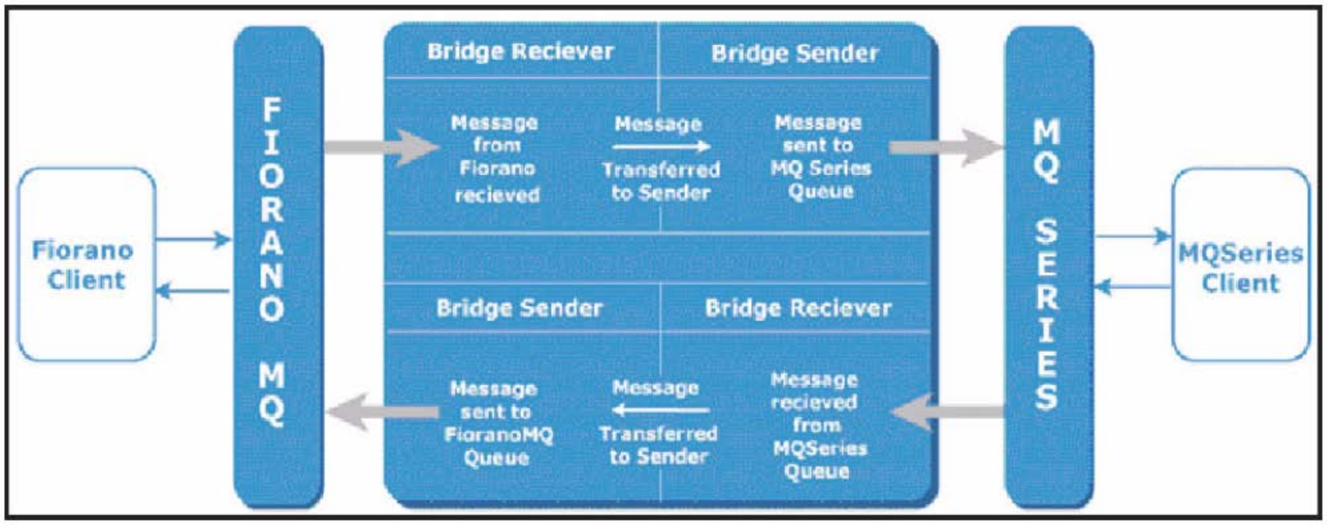
FioranoMQ Bridge solves this integration problem by allowing messages to be passed between FioranoMQ and other message queuing systems. If two banks were to merge and needed to integrate their information systems, the management can decide to consolidate all account information on IBM systems that use IBM WebSphereMQ for message exchange. The management would, however, like to have its customers continue using its current ATM system. The ATM system receives requests for account information and dispatches requests to the server system using FioranoMQ messages. Instead of rewriting applications, the FioranoMQ Bridge can be used to forward requests and responses between the two different message queuing systems.

Bridge Architecture [

The FioranoMQ Bridge is an open, standards based, Java component. The FioranoMQ Bridge provides a set of standards-based configurable services that allow messages to be exchanged between FioranoMQ and other message queuing systems.

Communication to any other JMS vendor is done using the standard JMS API. Communication to MSMQ is done using MSMQ Java APIs and communication to TIBCO Rendevous (Tibrv) is done using Tibrv Java APIs. The FioranoMQ Bridge sends messages as JMS Messages to the targeted JMS vendor. The FioranoMQ Bridge is configured using XML-based configuration files that allow a single instance of the Bridge to "bridge" any number of messaging servers.

Messages are pushed to the remote system using the standard JMS API (or MSMQ/Tibrv APIs if the remote system is MSMQ/Tibrv). Additionally, the Bridge creates receivers to asynchronously receive messages from the remote system. The Figure below depicts the flow between a FioranoMQ client and an IBM WebSphereMQ client through the FioranoMQ Bridge.



Forwarding Messages to Remote Queues

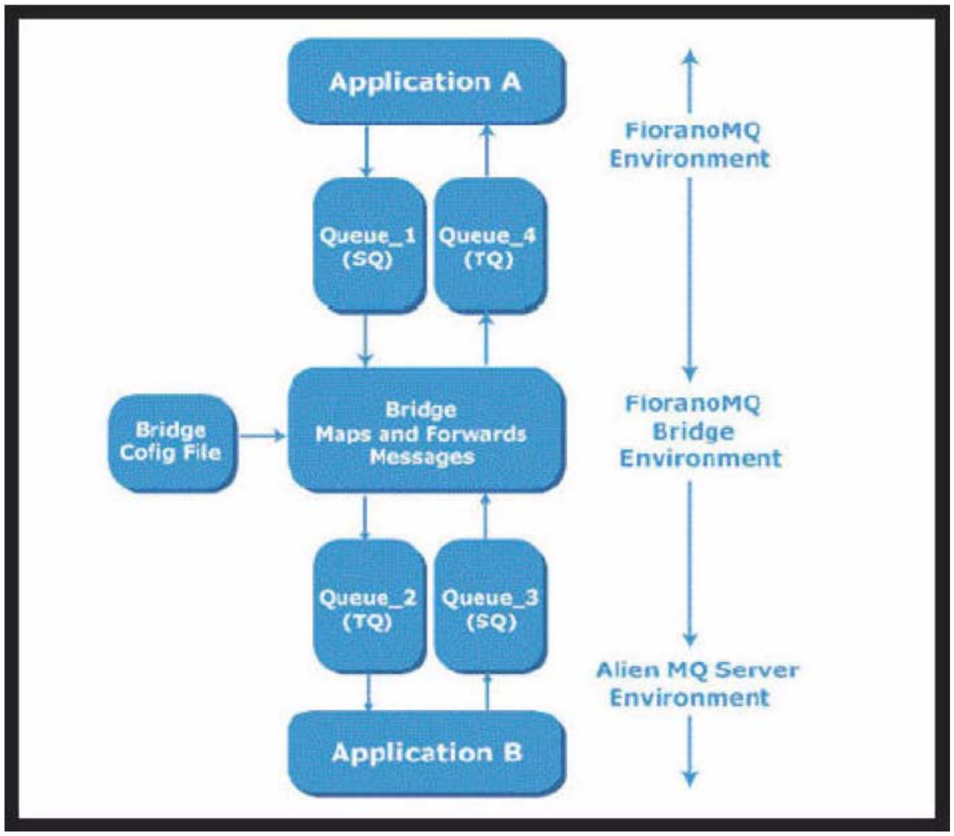
The FioranoMQ Bridge provides communication services between FioranoMQ and other messaging servers. As shown in Figure above, the Bridges create a relationship between a remote messaging server and FioranoMQ queues in order to send and receive messages.

This relationship is based on the use of two Bridges-specific entities, referred to as the Source Queue (SQ) and Target Queue (TQ):

Source Queue (SQ): An application always sends to a SQ, which is defined in the source messaging system. Messages received on a SQ are read by the Bridge and forwarded to the associated TQ. A SQ can either be a FioranoMQ queue or a queue on another messaging server ("subject" if Tibrv).

Target Queue (TQ): This is the final target queue for sending a message. This queue is defined by the remote messaging system. A Bridge sends messages (retrieved from the associated SQ) to this queue for subsequent processing by the target application. A TQ can be either a FioranoMQ queue or a queue on another messaging server ("subject" if Tibrv). The Figure below shows how the Bridge forwards a message from a FioranoMQ SQ to its associated Remote TQ.

For example, Application A is an ATM application that uses FioranoMQ for message exchange. It is designed to receive requests for account inquiries, send these requests to Application B, and return account information to bank customers. Application B is an account lookup application that uses IBM WebSphereMQ for message exchange.



Application A places an account inquiry message on Queue_1, a FioranoMQ queue. The part of FioranoMQ - WebSphereMQ Bridge reads the message on Queue_1. It maps the message header data into IBM WebSphereMQ format and forwards the message to Queue_2, an IBM WebSphereMQ queue. Application B reads the message on Queue_2, looks up the requested account information, and places the account information in a reply message. The message is placed on Queue_3, an IBM WebSphereMQ queue. The FioranoMQ Bridge reads the message on Queue_3. It maps the message header data into Fiorano MessageQ format and forwards the message to Queue_4, a FioranoMQ queue. Application A reads the message on Queue_4 and displays the account information to the customer.

Bridge Features

- **XML based**: The FioranoMQ administrator can configure the FioranoMQ Bridge using an XML config file. FioranoMQ Bridge gives the enterprise administrator complete power to configure message queuing servers on any network topology. The administrator can set up queues that need to be mapped between target and source servers. Configuring the Bridges is simple and is based on XML.

- **Robustness and Network Failures**: The Bridge has a built in feature to reconnect automatically with configured messaging servers. If a network connection or a messaging server goes down, the Bridge automatically tries to reconnect to the "down" servers after "configurable" periods of time. This pinging operation continues until a connection is re-established (or when the "down" server finally comes up again).
- **Avoiding Loopback in Bridges**: Cyclic links have been enabled in bridges. A FioranoMQ to FioranoMQ Bridge can result in an infinite loop being set up because of bridge configuration. A check has been added in the bridge which prevents an infinite loop. This check is enforced only if a particular configuration flag is turned on via Fiorano Studio.
- **Logging and Tracing Options**: The FioranoMQ Bridge provides comprehensive logging facilities. Logs can be redirected to files or to the console. The administrator can install "customized" logging mechanisms. Trace levels for the Bridge components can be set via Fiorano Studio.

Bridge Configuration

Configuring the Bridge consists of adding links and channels to the Bridge and specifying queue information to allow bi-directional communication across messaging servers.

When in offline mode, the administrator can easily add links to the bridge and configure source and target servers for message replication. Cluster administrators are provided with a template configuration file. FioranoMQ installation provides a default `ConfigswithBridges.xml` file in the default FioranoMQ profile (`fmq\server\profiles\FioranoMQ 9\conf` directory of MQ installation). This provides the default Bridge configuration, adding two links to link the source and the target servers bi-directionally. This file needs to be renamed as `configs.xml` prior to starting Fiorano Studio in the offline mode. The Fiorano Studio tool displays the Bridge with the default links that can be configured through the offline mode.

The following table lists configurable Bridge Manager properties:

Property Name Description

Parameters	**Description**
Avoid Loopback	This flag is used to avoid the loopback condition in bridge. The default value is True.
Name	Specifies the name of the Bridge
PingIntervel	Specifies the time interval after which the Bridge pings the source and target servers continuously to reconnect once a connection gets broken

Link Properties

The Bridge sends messages in the link specified between a source and a target server. A Bridge can have N number of links configured. By default, the server sets up only a single link to the Bridge. Properties related to the default link can be edited before creating and managing additional links or channels in the online mode. The Link element within the Bridge Manager MBean contains the following elements:

SourceServer

Specifies the server on which subscriptions are created. The Source Server element contains the ConnectionInfo.

TargetServer

Specifies the server on which publishers are created. Target Server contains the ConnectionInfo.

Connection Info Properties

The information present in the properties enables the Bridge to connect to servers even if they are behind Proxies or Firewalls. It enables secure connections by providing security certificates and a security protocol to be used. Connection Info comprises of the ServerURL, UserName, Password, ProxyURL, and ServerSecurityManager. The 'type' property of this element indicates the protocol over which the Bridge will connect to the source or target server.

ConnectionInfo contains the following elements:

Parameter	**Description**
ServerType	Type of the messaging server to connect to. Currently only four server types are supported: JMS, MSMQ, IBM WebSphereMQ, and Tibrv.
Parameter	These contain the different environment parameters required by the connector to connect to the Source/Target server. An important use of these parameters is to configure the protocol and the connection factory over which the Bridge will establish connections with the source and the target servers.

A channel can now be added to the existing link. The channel specifies the source and target queue information. A channel has the following information:

Parameters	**Description**
Name	Name of the Channel
SourceQueue	Specifies the source queue information required by the Bridge. This contains the Source Queue name.
TargetQueue	Specifies the target queue information of the channel. This contains the Target Queue name.
Parameter	These contain the different environment parameters required by the connector to connect to the Source/Target server. An important use of these parameters is to configure the protocol and the connection factory over which the Bridge will establish connections with the source and the target servers

Chapter 15: Large Message Support

With Large Message Support (LMS) in FioranoMQ, clients can transfer large messages in the form of files with, theoretically, no limit on the message size. Large messages can be attached with any JMS message and the client can be sure of a reliable and secure transfer of the message through FioranoMQ Server.

Note: In this chapter, the terms "Large Message Support" and "LMS" is used interchangeably.

Salient Features

Reliable transfer of large messages

The Large message sender and receiver use the JMS semantics to transfer arbitrarily large messages over the network in chunks, using reliable JMS semantics. When large message transfer is initiated, all message fragments are sent through the server ensuring reliability. Files sizes in Gigabytes can be transferred.

No increase in cache/JVM heap size required

Although the message transfer happens through the server, the file fragments are sent to the receiver in a request-reply fashion. This enables the server to handle large message transfers without any increase in the cache/JVM heap size.

The Large message transfer is not restricted to any queue or topic

As mentioned above, large message transfer conforms to JMS semantics, allowing the user to transfer the message on any queue or topic.

Resume function at both sender and receiver end]

Broken transfers can be restored using the resume functionality built into the message class.

Message transfers are specific to the JMS Connection-Users involved in the transfer. The user can resume the transfers at any later point provided the participant at the other end is available.

Minimal changes in the application code

All the message transfer properties are built into the message object while the message transfer semantics are built into the normal JMS send and receive calls. Therefore, only minimal code changes are needed to send and receive large sized data.

Using Fiorano LMS to Transfer Large Files

A large message is defined as a JMS Message with a reference to a large sized file. Message fragmentation, reassembly, sequencing, duplication, and recovery are handled internally. When the large message is sent, it is only the reference to the large file that is sent with the JMS Message. The actual file transfer happens only after the receiver has received the JMS message containing the reference to the message and starts the saveTo operation on it.

Transferring a large message using Fiorano LMS involves the following phases:

- Message creation
- Starting the message transfer
- Tracking the message transfer
- Handling exceptions in the message transfer
- Resuming the message transfer

Message Creation

The large message is created using the complete JMS semantics. A JMS application can specify a string property; "JMSX_LM_PATH" to convert a JMS message to a large message. This string property specifies the absolute path of the large file that needs to be sent to the listening consumers. A consumer receives the large message just like any other JMS message.

Starting the Message Transfer

The large message transfer can be initialized by calling the normal JMS send call at the producer end. This posts the large message to the intended destination (queue/topic), which is received by the listening consumers. They receive this large message and can start the actual transfer of the large file by opting to save this message at the required location using the saveTo API.

Tracking the Message Transfer

Message transfer can be tracked asynchronously by registering the status listener with every large message. The status listener can be set using the `setLMStatusListener` API.

Handling Exceptions During Message Transfer

Exceptions can be handled using the status listener registered with the large message. When an exception occurs while transferring the large message, the application is notified via the registered status listener. Applications can check the type of error and status of the transfer and can decide to resume/cancel the message transfer.

Resuming the Message Transfer

If a failure occurs while transferring the message, the application can resume the message transfer from the state at which the failure had occurred. The application can resume the message transfer from the two levels described below:

- **Resume on exception**: FioranoMQ notifies the application about the status of the message transfer using the registered status listener. If an exception occurs during message transfer, FioranoMQ notifies the application about the failure. Upon failure notification, the application can opt to resume the message transfer. It can resume the transfer using the `resumeSend ()` or `resumeSaveTo ()` APIs provided in the message object.
- **Resume on startup**: The JMS Connection object keeps track of all active message transfers fora particular connection. If the application becomes unavailable (JVM down) while participating in the message transfer, it can then check the list of unfinished transfers from the connection object. The connection object provides a list of unfinished messages required to be sent and received. The application can resume the transfer using the `resumeSend ()` or `resumeSaveTo ()` APIs provided in the message object.

Salient Features

The actual transfer of the large file attached with the large message follows certain message transfer semantics explained below:

Consumer Discovery

The message producer waits for the initial message (handshake) from the consumer before any message fragments are sent. In the point-to-point (request/reply) messaging model, the discovery phase ends as soon as the handshake message is received from the consumer. In the publish-subscribe model, the discovery phase continues to receive handshake messages from new consumers. The duration for which the producer waits for handshake messages is determined by the `requestTimeOut` value set by the user in the message object.

Fragment Size

A large file is broken into fragments of size <fragment size> as set by the user in the message object. Fragments in multiples of TCP/IP window size are recommended for optimal performance.

Window Size

Window size is the number of message fragments sent by the message producer before an acknowledgment is received from the message consumer. For instance, for a window size of 50, 50 fragments of size <fragment size> are sent to the server. The producer then stops sending new fragments until an acknowledgment is received from the message consumer. Once an acknowledgment is received, the message transfer continues.

Sequencing

Every message fragment sent has a sequence number attached to it by the message producer. The message consumer expects the fragments to be received in the same sequence as the order in which they were sent. If the sequence is disturbed then the consumer re-requests those fragments that could not be received sequentially. A producer, on receiving such requests, starts sending the fragments from the sequence number mentioned in the request.

Handling Duplication

In rare cases a message fragment is received more than once. In such cases, the message consumer ignores the duplicate fragments.

Handling lost fragments

The message consumer identifies lost fragments using the sequence number associated with each fragment. When a fragment is not received, the consumer re-requests those fragments which could not be received sequentially. A producer, upon receiving such requests, starts sending the fragments from the sequence number mentioned in the request.

Optimizing Large Message Transfer

The time involved in transferring a large message (of size M) is the sum of: [

- The time involved in transferring N JMS Messages of size M/N.
- Time involved in I/O operations (reading the large file on the sender side + saving the large file at the receiver end)
- The time involved in establishing the connection and ensuring fragment sequence and reliability.

The following factors can affect the performance of LMS:

Fragment Size

A small fragment size implies frequent I/O operations (reading the source file and saving the target file) as well as frequent socket calls to send the fragments to the server. Large fragment size implies increased memory usage both at the client and server ends. The optimal fragment size should be small enough to be held in the memory of the sender, server and receiver. Fragment sizes, whether small or large, should be in multiples of TCP/IP packet size for deriving optimal performance.

Window Size

A large window size implies increased memory usage at the server end because of the possibility of the receiver being slow to receive the messages from the server. After a window of fragments is sent, the message producer halts itself to receive an acknowledgment from the server. The receiver can not receive a fragment sequentially and request the fragment to be sent again. It is wise to have smaller window sizes when the expected rate of fragment loss is high. Another factor to be considered is the latency between the sender and the receiver. It is better to have a large window size when the latency is high.

Status message frequency

The `setLMStatusListener` call registers a status listener with the message transfer and informs the user application of the status of the message transfer. It also takes another parameter for the frequency of status updates required by the user. The value of this parameter is the number of windows of message fragments sent before the user application is updated to the status of 'transfer'. A small value (say 1, for an update after every window) hampers the message transfer rate considerably. Performing intensive operations in the `onLMStatus` call takes up considerable time by the message transfer thread.

Known Limitations

- Only one file can be associated with a JMS message.
- Large messages cannot be sent in a transacted session.
- Resume API does not work well at times in cases of publish-subscribe.
- Resume directory cannot be shared by multiple consumers receiving a large message.
- LMS does not work if `AllowDurableConnections` is set to TRUE
- LMS does not work if the MQ Servers are run in High Availability (HA) mode.
- An instance of a large message cannot be sent again unless the previous transfer is complete.
- LMS works only for unified connections provided by the JMS 1.1 specifications.

Chapter 16: High Availability

Today's real-time enterprise solutions often deploy messaging middleware that enables communication between various sub components. This middleware is entrusted with important data that should be delivered reliably and as fast as possible to the recipient application. The middleware server might also be required to store this data in its data store until it is delivered.

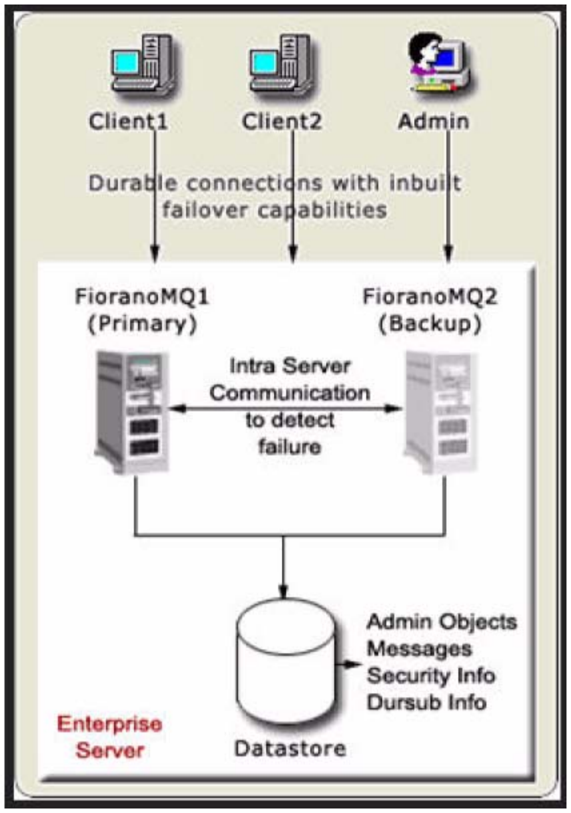
A failure in the middleware message bus can bring the entire system to its knees within seconds. It is absolutely imperative for the messaging backbone to provide a backup facility that allows messaging operations to resume quickly in the event of a failure. This backup server restores the state of the original message server to the state prior to failure. Any data stored in the server's data store is accessible through the backup server. Switching from one server to its back up should be automatic and transparent to the client application.

FioranoMQ implements High Availability (HA), which allows JMS applications to take advantage of in-built fault tolerance capabilities. This chapter discusses the salient features of FioranoMQ's HA solution. It explains the functions and underlying architecture of the entire solution together with step-by-step instructions for enabling HA in FioranoMQ.

FioranoMQ's HA - An overview

FioranoMQ servers running in HA mode have a designated backup server that is started, together, with the primary FioranoMQ server. If the primary server becomes unavailable due to any reason; the backup server picks up messaging traffic immediately. This pair of primary and its backup server is known as an Enterprise Server and this term is used to describe this primary/backup pair throughout the document.

This Enterprise Server represents a HA entity that appears as a single FioranoMQ server to its applications. JMS applications, during initialization, connect to the primary FioranoMQ server, if available. If the primary server goes down due to any reason, all connections are automatically routed to the backup server and communications are restored immediately. Since this is transparent to the client application, the client application need not worry about the reconnection logic in its code as these are handled by FioranoMQ's client runtime library internally.



HA Components

The section below describes the components and associated concepts that collectively form an Enterprise Server.

Backup Server

Fiorano's HA solution requires running a backup FioranoMQ server. This server (also referred to as secondary server in this document) can be started on the same or on a different machine. The server picks up all the messaging traffic as soon as it detects the unavailability of its Primary server.

Server States

FioranoMQ Servers that make up the Enterprise Server can be in either active or passive state. 'Active state' refers to the normal working mode of the server. In 'passive mode', the server only monitors its peer and does not handle any client requests. Client connections to a server in passive mode are refused. On startup, the server establishes communication with its peer server. If this peer server is 'alive', the current server enters passive mode. It leaves the passive mode and becomes active (accepts client connections) only when it detects that the peer is unavailable.

Intra-Enterprise Server Communication

Both Primary and Secondary servers open and listen on a TCP/IP port that allows them to establish a dedicated connection between themselves. This port is different from the one used by client applications for connecting to the server and hence does not affect normal messaging operations in the active server. A connection is established during the initialization phase of the servers and is used for exchanging 'health' and 'state' information between the two servers. Information gathered is used by the servers to switch to 'active' from 'passive' state (and vice-versa), if and when required.

Common Persistent Message Store

Both Primary and Secondary servers are required to logically access a common persistent message store. To achieve this, the FioranoMQ administrator can either point both the servers to the same physical database or can set up replication between the databases of the two servers. Both these options are available with Fiorano's file-based database as well as on third party JDBC compliant RDBMS Servers.

Common Admin and Security

Besides the message store, the Primary and Backup Server in an enterprise server must share the admin objects (Destinations and Connection factories) and Security Information (ACLs and User Information) amongst themselves. This is achieved by using a common naming and realm storage (like RDBMS and LDAP) or setting up replication on these databases between the two servers.

Gateway Machine

Consider a scenario where the enterprise server consists of FioranoMQ server 1 and FioranoMQ server 2. Both these servers are constantly monitoring each other's status without any problems. Now, assume FioranoMQ Server 2 exits the network due to some network failure. Though FioranoMQ Server 2 is still running, it is no longer connected to the network (and hence not accessible to FioranoMQ Server 2 and to related client applications). In this scenario, a third gateway machine is used to detect the HA server which is no longer available on the network. It becomes imperative to choose a gateway machine that is least expected to fall out of the network. It is best to use the actual gateway server of the network, in which the enterprise server is deployed, as the Gateway machine for HA.

Note: If the Gateway machine was to exit the network, HA continues to function properly as long as the two HA servers are present on the network. If one of the HA servers also goes out of the network, then it is not possible to reach a consistent state. In such a situation, both the servers' switches to passive mode and the enterprise server will not be in a position to process client requests. However, the enterprise server would be available for client requests when either both peers are up or if one of the peers and the gateway machine are available on the network.

FioranoMQ HA Salient Features

Shared and Replication of Databases

FioranoMQ provides complete flexibility to administrators giving them an option of either using a shared database (between active and passive servers) or using database replication (from active to passive server). “Shared” HA typically provides much better performance in comparison with “Replicated” HA. If it is not possible to share a database, administrators can still use FioranoMQ's HA using inbuilt replication support.

Application Failover

If the primary server becomes unavailable, all the client applications connected to it are automatically re-connected to the secondary server. The process of shifting from the primary server to the backup server or vice-versa is transparent to the application. The application does not need to implement any reconnect logic in its code. Re-connection achieved by connecting to the server through a Durable Connection. If a backup server is available, the Durable Connection will connect to the backup server. Otherwise it waits for the server to restart and during the disconnected period stores all data in to a local repository. This data is re-transferred to the server as soon as a connection is re-established, making the system highly reliable and robust even in the event of network failures.

Note: Durable connections implement ‘client side persistence’ and are a proprietary feature of FioranoMQ (though it does not require any proprietary APIs) and should not be confused with Durable Subscribers.

Data Store Consistency (maintained between server switches)

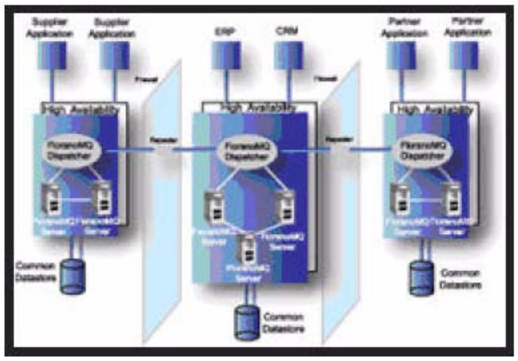
When the primary server becomes unavailable, its backend database state is preserved. This state is picked up by the secondary server when it becomes available. This avoids loss of persistent information between server switches while, at the same time, providing access to information stored to the backup server. For example, all the messages published on various destinations residing on the primary server are available to valid consumers through the secondary backup server without loss.

Expensive HA Hardware Not Required

Fiorano's HA solution is implemented using software and is not dependent on expensive hardware solutions. It can run on any java-supported platform. With the shared database option, one might want to use RAID or SAN disks (if using HA over Fiorano's proprietary file-based data store) for enhanced speed and stability, but this hardware is not necessary for Fiorano's HA solution. Using either replication support or using a central RDBMS server as the message store in the Enterprise Server avoids the need for additional hardware.

Implementing a Cluster

The Enterprise Server can be clustered with other Enterprise Servers or even stand-alone FioranoMQ servers. The cluster can share destinations (using a common naming store) and provide load-balancing facilities.



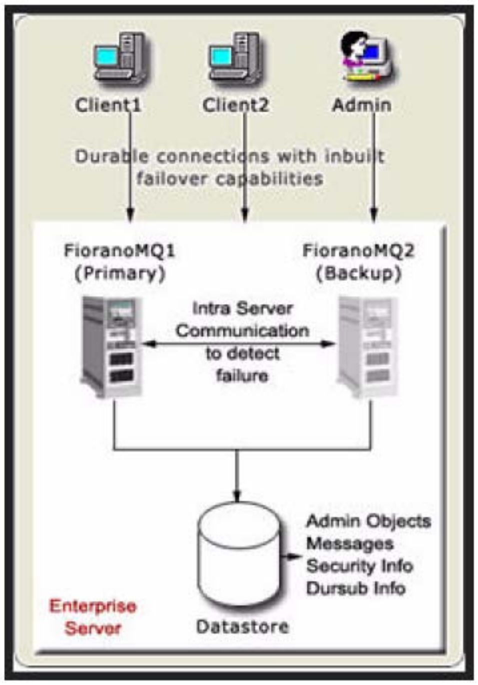
HA Example Scenario

This section of the document provides a description of events that occur if of the servers in the Enterprise Server becomes unavailable.

State - 1 (Normal Operation State)

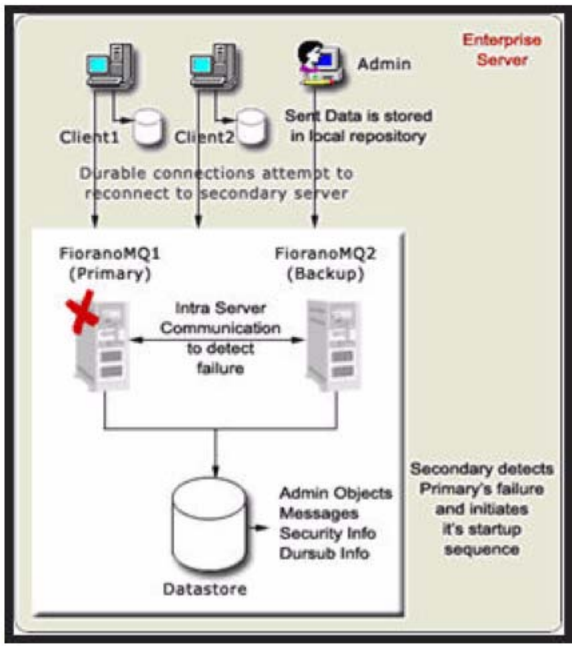
All client applications are connected to one of the servers in the Enterprise Server. It can be assumed that the clients are connected to the primary server at this instant. The backup server is up and running in passive mode. The backup server will not accept any client connections at this point of time. Primary and Backup servers are continuously exchanging health information over a dedicated channel. All persistent information is being stored in the 'back-end' data store through the primary server.

The above scenario is illustrated below:



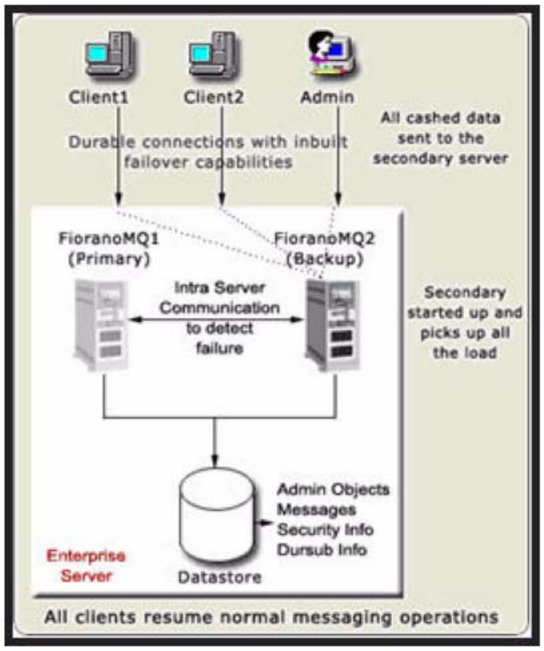
State - 2 (Active Server goes down)

The backup server detects the primary server's failure and initiates its start up sequence. All client applications connected to the primary server detect the problem and Fiorano's runtime library internally attempts to re-connect to the secondary server. New messages published during this downtime are stored in a local repository in all client machines.


Enterprise Server
Admin
Client1
Client2
Sent Data is stored in local repository
Durable connections attempt to reconnect to secondary server
FioranoMQ1 (Primary)
FioranoMQ2 (Backup)
Intra Server Communication to detect failure
Admin Objects
Messages
Security Info
Dursub Info
Datastore
Secondary detects Primary's failure and initiates it's startup sequence

State - 3 (Backup Server resumes operations)

The backup Server then starts up and gets ready to take up client connections. All client applications reconnect with the secondary server once the backup server is up, such re-connections being managed automatically by the FioranoMQ client runtime system. Messages stored in the local repository (that were sent during down-time) are sent to the secondary server. All durable consumers continue to pick up messages from where they had left off.


Client1
Client2
Admin
Durable connections with inbuilt failover capabilities
All cashed data sent to the secondary server
FioranoMQ1 (Primary)
FioranoMQ2 (Backup)
Intra Server Communication to detect failure
Secondary started up and picks up all the load
Admin Objects
Messages
Security Info
Dursub Info
Enterprise Server
Datastore
All clients resume normal messaging operations

When the primary server restarts, it detects that its backup is alive and enter into passive mode. The primary server continuously pings the backup server and initiates its startup sequence if the backup server goes down due to any reason. In this sense, the original backup is now the Primary and the original Primary is now the backup server.

Limitations of HA

Client level transactions do not span across servers in the Enterprise Server when running in shared mode. Transacted sessions involving receivers will be rolled back if the primary server crashes. Therefore, the messages delivered in that transaction are redelivered to the receivers upon connection to the backup server.

Distributed transactions, which are in execution during the transition phase, become "in-doubt transactions" when the primary server goes down. These transactions get ‘rolled back’ and can be recovered after the client reconnects to the secondary server.

JMS Topic Requestor can not receive its intended reply if failover occurs after a request is sent. This occurs because JMS Topic Requestor creates a non-durable subscriber, which can miss a message during failover. However, if a topic requestor creates a durable subscriber to listen for replies, then it works even during failover.

If both HA servers (primary as well as backup) go down, the requestor receives a duplicate reply (with `reDelivered` Flag = true) for the first request made after failover occurs.

Chapter 17: Distributed Transactions

Communication between diverse applications is an essential requirement in today's distributed network environments. This communication can be at a single tier where one application communicates to another in a single transaction, or at a multiple tiers where many applications communicating to each other in one large transaction. The ability of a transaction to span multiple applications is fundamental to multi-process communication.

The availability of low-cost computing power and increased network bandwidth gives rise to distributed component-based computing applications. Distributed computing applications and distributed transactions are vital for developing computing components for multi-tier applications, which can run on different platforms or through networks. Here, a transaction is defined as a unit of work composed of sets of operations on objects. A distributed component-based application is a configuration of services provided by different application components. These components are executed on physically independent systems running on multiple machines. To the user, these components appear as a single application running on a single physical machine.

Introduction

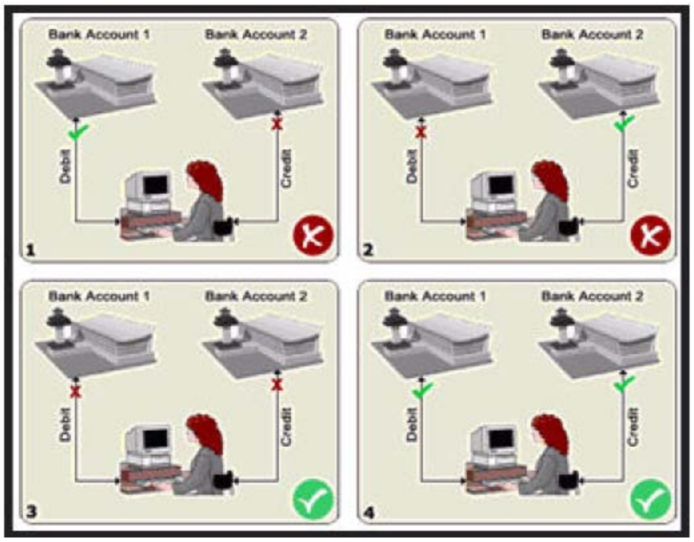
In the world of distributed computing and distributed transactions, a transaction can be defined as a group of statements representing a unit of work composed of a set of operations on objects. These groups of statements must be executed in totality as a unit. Transactions can also be viewed as sequences of operations on resources, such as read, write, or update, which transform one consistent state of the system into a new consistent state. The basis of these transactions lies in the fundamental concept of an all-or-nothing proposition. Either all steps of a transaction must be completed successfully or none of them should be completed.

In a large network, spread over multiple machines and involving many individual steps for a transaction, it is highly probable that one of these steps do not get completed. This can occur due to many reasons such as flawed application logic, server failure, hardware failure, and network interruptions. Due to unpredictable environmental factors, transactions must adhere to the following properties:

- **Atomicity**: This is the all-or-nothing property. Either the entire sequence of operations is successful or unsuccessful. A transaction should be treated as a single unit of operation. Only completed transactions are committed and incomplete transactions are rolled back or restored to their original state.
- **Consistency**: A transaction maps one consistent state of resources (for example, database) to another. Consistency is concerned with correctly reflecting the reality of the state of the resources. Some examples of consistency are referential integrity of the database and unique primary keys in tables.
- **Isolation**: A transaction should not reveal its results to other concurrent transactions before it commits. Isolation assures that transactions do not access data that is being concurrently updated. The other name for isolation is serialization.
- **Durability**: Results of completed transactions have to be made permanent and cannot be erased from the database if the system fails. Resource managers ensure that the results of a transaction are not altered due to system failures.

Use Case

A familiar example of distributed transactions is transferring money from one bank account to another. The transfer comprises of two separate actions involving debit of a certain sum of money, and the credit of this sum to another account. Both these steps will be a part of a single transaction. Both steps must be completed for the completion of the entire transaction. If only one of the two steps gets completed, the following possible problems are likely to arise:



Case 1 and 2 are summarized in the following table.

Step 1	**Step 2**	**Result**
Completed Successfully	Incomplete connection failure	The amount has been debited from the customer's account but the payment has not been received. This causes the customer to lose money.
Payment has not been made by the customer	The amount is credited to the sellers account	The bank loses money.

In the above example, a single transaction should ensure that the either-or-none proposition holds true. Transactions do not always involve the transfer of funds as in the banking example just covered. Transactions are necessary for all kinds of business activities. For example, an online bookstore performs transactions that include ordering books from suppliers, transferring inventory from suppliers, updating available quantities of books accurately, charging customers appropriately for purchases and fulfilling customer orders. All of these actions, and a multitude of others, can need to be executed within a single transaction.

In other words, a transaction is a unit of work performed on behalf of a single client, delimiting a related set of operations and providing scope for concurrency. Transactions guarantee consistency of a 'shared state' in the occurrence of concurrent access by isolating one client's work from another and undoing a client's work in case of a failure.

Transactions and the Distributed Transaction Processing (DTP) System

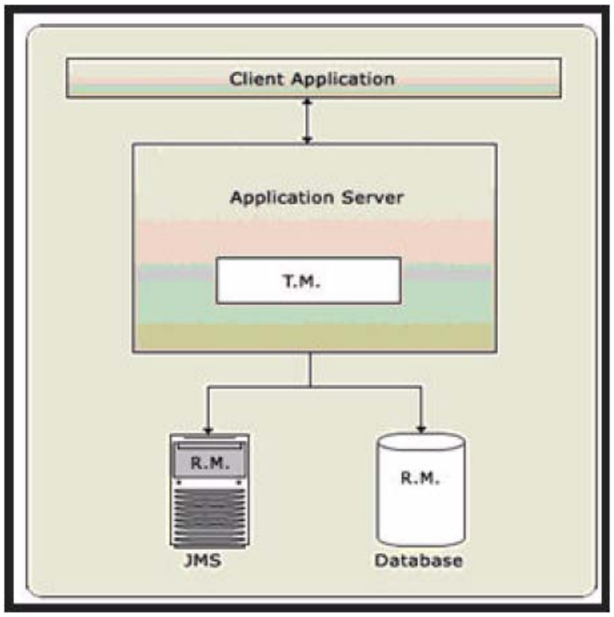
A Distributed Transaction Processing (DTP) system defines and coordinates interactions between multiple users and databases (or other shared resources) residing on multiple machines. When a transaction includes operations in several databases or other shared resources, the goal of a DTP system is to carry out this transaction in an efficient, reliable, and coordinated manner. The transaction must comply with ACID properties as far as possible.

Components of a Distributed Transaction

The transaction manager and the resource manager are the two key elements of any transactional system. Generally, distributed transactions have five components:

- Transaction manager
- Application server
- Resource manager
- Application program
- Communication resource manager

Each of these contributes to the distributed transaction processing system by implementing different sets of transaction APIs and functionalities.



- A Transaction Manager provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.
- An application server provides the infrastructure required to support the application runtime environment, which includes transaction state management. An example of such an application server is an EJB server.

- A Resource Manager (RM) provides the application access to resources through a resource adapter. The resource manager participates in distributed transactions by implementing a transaction resource interface. This interface is used by the transaction manager to communicate transaction associations, transaction completion as well as recovery work. FioranoMQ is a resource manager.
- A transactional application specifies actions that form part of a transaction. These programs can require actions such as updating a database or sending messages. Such standalone Java client programs might want to control their transaction boundaries using a high-level interface provided by an application server or the transaction manager.
- A Communication Resource Manager (CRM) supports transaction context propagation and access to the transaction service for incoming requests.

From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed. Only high-level interfaces need to be defined to allow transaction demarcation, resource enlistment, synchronization and recovery processes to be driven by the users of the transaction services.

FioranoMQ as a Distributed Transaction Resource Manager

The JMS specification provides a model that outlines how a messaging system should behave in a transactional environment. In the JMS transactional model, message producers and message consumers never participate in a single distributed transaction. Instead, the JMS specification has its own loosely coupled transaction model, whereby each message producer or message consumer has its own private transactional session with the messaging system, such as FioranoMQ.

This is due to the inherent nature of applications using messaging to communicate in an asynchronous environment. Senders and receivers of messages are abstractly decoupled from each other. A receiver can not be available at the time the sender initiates a transaction. Thus, the sender has a contract with the JMS messaging system, which ensures messages produced within a transactional session are "committed" to the JMS messaging system in an all-or-nothing manner. Likewise, the receiver has another contract with the JMS provider ensuring all messages being consumed within a transactional session are received in an all-or-nothing manner.

FioranoMQ, being s standards based JMS server, implements the above-mentioned model to provide XA support in a Distributed Transaction environment using the JMS XA Service Provider Interface (JMS XA SPI).

In addition to the high performance "file based" data store, FioranoMQ supports the use of Relational Database (like Oracle) as message data stores.

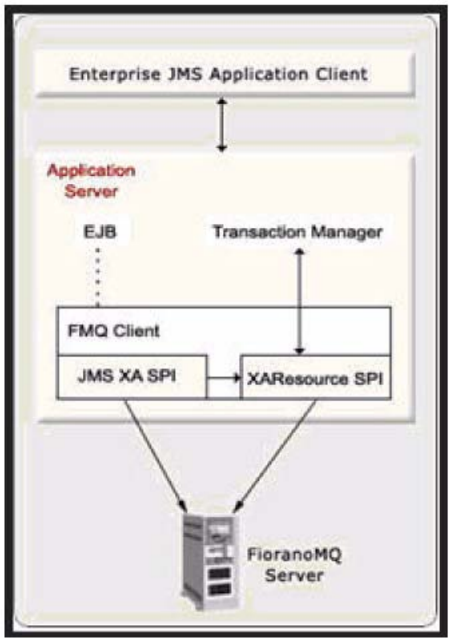
Transactions with J2EE

J2EE Applications include components that take advantage of infrastructural services provided by the J2EE Container and Server, and therefore need only to focus on "Business logic".

Transactional support is an important infrastructural service offered by the J2EE platform. The specification describes the Java Transaction API (JTA), whose major interfaces include: `javax.transaction`, `UserTransaction`, and the `javax.jms.TransactionManager`. The `UserTransaction` remains exposed to application components, while the underlying interaction between the J2EE server and the JTA `TransactionManager` is transparent to the application components. The JTA `UserTransaction` and JDBC's transactional support are both available to J2EE components.

An EJB (Enterprise Java Bean) running in an Application Server exposes the business logic methods that clients invoke to perform useful operations, such as depositing or withdrawing from a bank account. Enterprise beans are capable of handling transactions. This implies that EJBs can fully leverage the ACID properties to perform reliable, robust server side operations. Thus, EJBs are ideal modules for performing mission critical transactional tasks.

In an EJB, the code never interacts with the low level transactional system. The beans never interact with a transaction manager or a resource manager.



The application logic needs to be written at a high level, without regard for specific underlying transaction systems. The low-level transaction system is abstract to the EJB container that runs behind the scenes.

The bean components are responsible for deciding when a transaction should begin, commit or abort. If the system executes properly, a commit is invoked; else an abort is invoked.

It is the XAResource provider (FioranoMQ) responsibility to integrate itself seamlessly with the transaction in progress and allow for behind the scenes enlistment and delistment. Commit operations, with transaction propagation managed by the application server.

This provides a much higher level of abstraction as compared to using an external transaction manager. In addition, it achieves the objective of isolating the "business logic designer"/"bean designer" from the details of how the transaction progresses. The bean no longer has to take care of specific enlisting, delisting, and committing or rolling back transactions in progress. This is in contrast to controlling transactions using an external transaction Manager.

The `javax.transaction.UserTransaction` interface defines methods that allow applications to define transaction boundaries and explicitly manage transactions. The `UserTransaction` implementation provides application components -- servlets, JSPs, EJBs (with bean-managed transactions) -- with the ability to programmatically control transaction boundaries. EJB components can access `UserTransaction` through EJBContext using the `getUserTransaction ()` method. Some of the methods specified in the UserTransaction interface include `begin ()`, `commit ()`, `getStatus ()`, `rollback ()` and `setRollbackOnly ()`. The J2EE server provides the object that implements the `javax.transaction.UserTransaction` interface and makes it available through JNDI lookup.

FioranoMQ as a XAResource provider can be integrated with any J2EE application server so as to perform in a predictable manner when used with a `UserTransaction` object provided by the application server.

FioranoMQ XA Implementation Notes

Distributed transactions are supported for topics as well as queues as long as the data store used is able to participate in distributed transactions. FioranoMQ supports the use of RDBMS as the data store for messages which needs to be used in XA scenarios. Fiorano's File based data store cannot be used for destinations that need to be used in XA.

FioranoMQ XA implementation does provide support for local transactions as well. This implies that when a new XA session is created, it has a local transaction context. This local transaction context ends when the XA session gets associated with a global transaction context. XAResource associated with the session starts a new distributed transaction (XAResource.start). When the global transaction ends - (XAResource.end() or XAResource.rollback()) - xasession automatically switches back to the local transaction context where the JMS session of the xa session can be used as a transacted JMS session, invoking its commit/rollback methods.

The behavior of createSession call on the XAConnection object is undefined in the JTA specifications.

Session session = `xaConnection.createSession(boolean transacted, int ackMode);`

- **boolean transacted**: usage undefined.
- **int ackmode**: usage undefined.

FioranoMQ behaves in the following way when it encounters a createSession (or createTopicSession/createQueueSession) on the xaConnection (or XATopicConnection/XAQueueConnection) object:

It returns the session (or topicsession/queueSession) object whose behavior is defined entirelyby the specified parameters. For example, in the following call a transacted (non-xa) session object is returned, which can take part in the JMS transactions.

```
xaConnection.createSession(true, 0);
```

All transactions are rolled back on server startup or on application close.

If a FioranoMQ client terminates after a transaction has successfully ended, (`XAResource.TMSuccess`)/suspended (`XAResource.TMSUSPEND`)/ prepared (returned with a `XAResource.XA_OK` flag), then this transaction can be used from the same status (prepared, ended or suspended), upon client re-activation. Similarly, if a FioranoMQ Server terminates when any ended/suspended/prepared transactions are active then, upon restart, FioranoMQ clients remain unaffected and can continue working on the transaction.

RDBMS based topics, on which messages are published in a transaction get locked in the 'prepare' call. No other transaction can be prepared in which messages published on locked topics. All Non-xa publishers publishing on a locked topic have to wait for the release of the lock to publish a message.

Limitations of XA Implementation of FioranoMQ

XAResource's Transaction `timeout()` APIs and `forget()` APIs are not supported in this release.

A durable subscriber can be associated with only one distributed transaction at any point of time. It can be associated with the next transaction only when it is disassociated with the previous transaction. The durable subscriber gets associated with the transaction when the xaResource object, created from the same session object as the durableSubscriber, starts a new transaction. This association ends when the resource object performs a commitrollback on the transaction that is started.

The following code snippet displays the point at which the association of the durable subscriber ends with the transaction.

```
// Get the topic session object

TopicSession ts = xats.getTopicSession();

// create a durable subscriber. This subscriber is not associated with

// the distributed transaction as the distributed transaction has not started yet.

TopicSubscriber   subscriber = ts.createDurableSubscriber(topic,"clientId");

// get the xaResource object

XAResource xaresource = xats.getXAResource();

// start the resource Object. During this call, above durable subscriber gets

// associated with the transaction.

xaresource.start(xid,XAResource.TMNOFLAGS);
```

Applications perform some work here:

// end the resource object. The association does not end when the resource ends.

```
xaresource.end(xid,XAResource.TMSUCCESS);

// prepare the resource object
```

```
xaresource.prepare(xid);

// commit the resource object. The association ends when the resource object gets

// committed. The association ends when the resource objects rollbacks.

xaresource.commit(xid, false);
```

The methods with which a durable subscriber can be disassociated from the distributed transaction are:

Single phase commit: Committing the transaction in a single phase disassociates the durable subscriber from the distributed transaction.

```
        xaresource.commit(xid, true);
```

Two phase Commit: Committing the transaction in two phases disassociates the durable subscriber from the distributed transaction.

```
        xaresource.prepare(xid);

        xaresource.commit(xid, false);
```

Rolling back the transaction: Rolling back the transaction disassociates the durable subscriber from the distributed transaction.

```
        xaresource.rollback(xid);
```

Invalid attempts: The association of the durable subscriber does not end when the resource object ends the distributed transaction. As explained above, the association ends only when the transactions commit/rollbacks the transaction that has started. If a durable subscriber is associated with a distributed transaction that has ended, performing any of the following functions leads to the throwing of an exception by the FioranoMQ server.

Start transaction: FioranoMQ Server throws an exception in case an attempt is made to start a new transaction when the previously started transaction has not committed/rolled back yet.

The following code snippet displays the above:

```
// create a durable subscriber.

// TopicSubscriber   subscriber =  ts.createDurableSubscriber(topic,"clientId");

// start a resource Object.

xaresource.start(xid,XAResource.TMNOFLAGS);

 ..

// end the resource object.

xaresource.end(xid,XAResource.TMSUCCESS);

// Attempt to start a different transaction leads to an exception

xaresource.start(xid2,XAResource.TMNOFLAGS);
```

The application provider is advised to commit or rollback the transactions that have ended, before starting new ones.

The following code snippet explains the work-around to the mentioned problem:

```
// get the resource object

xaresource = xats.getResource();
```

The transaction should be committed before creating the durable subscriber. The transaction can be committed either in a single phase or in two phases.

```
xaresource.prepare(xid);

xaresource.commit(xid,false);
```

or

```
xaresource.commit(xid,true);
```

If the transaction cannot be committed, the transaction should be rolled back before creating the durable subscriber.

```
xaresource.rollback(xid);
```

Restart the application: FioranoMQ retains the state of all the transactions that have ended or that are prepared. If an application crashes after ending a transaction, it can restart/prepare/commit/rollback the transaction. If an application, where a durable subscriber is associated with the ended transaction, crashes, any attempt to use the subscriber in non-xa transactions or in any other transaction leads to an exception.

An example: A durable subscriber is associated with a distributed transaction; xid. The application crashes after ending the transaction xid. When the application restarts, the following sequence leads to an exception:

Subscriber creation without starting the transaction: An attempt to create a subscriber, without starting the transaction those ending leads to an exception thrown by the JMS server.

```
// get the resource object

xaresource = xats.getResource();

// attempting to create a durable subscriber, without starting the transaction leads
to

//an exception.

dsubscriber = ts.createDurableSubscriber(topic, "subId");
```

It is recommended that the application provider should commit/rollback/restart the ended distributed transaction before creating the durable subscriber.

```
// get the resource object

xaresource = xats.getResource();
```

The transaction should be committed before creating the durable subscriber. The transaction can be committed either in a single phase or in two phases.

```
xaresource.prepare(xid);
```

```
xaresource.commit(xid,false);
```

Or

```
xaresource.commit(xid,true);
```

If the transaction cannot be committed, the transaction should be rolled back before creating the durable subscriber.

```
 xaresource.rollback(xid);
```

In case the transaction needs to be restarted, it should be restarted before creating the durable subscriber.

```
 xaresource.start(xid, XAResource.TMJOIN);
```

```
//  Creating the durable subscriber is possible after this: dsubscriber =
ts.createDurableSubscriber(topic, "subId");
```

```
// attempting to create a durable subscriber, without starting the transaction leads
to an exception.
```

```
dsubscriber = ts.createDurableSubscriber(topic, "subId");
```

Distributed Transactions do not work for unified connections.

Note: If a transaction is in the start phase, then it will rollback automatically when the server terminates.

Chapter 18: FioranoMQ Content Based Routing

Routing and addressing messages on communication networks is handled by specialized addressing and routing information present in the message header. Several applications, however, require a message to be addressed and routed according to the contents of the message. For XML messages, it is possible to perform content-based addressing and routing using X-Path predicates and SQL 92 syntax. FioranoMQ Content-based Routing (CBR) combines the world's fastest JMS server with ultra high-speed proprietary routing algorithms to provide fast content-based routing that is scalable.

FioranoMQ Content Based Routing

Fiorano is the first to introduce an intelligent, standards-based message, content-based routing system.

This chapter outlines the content-based routing (CBR) problem that FioranoMQ CBR solves, a description of how to use CBR with samples, includes a description of the types of XML currently supported, a section covering the entire subset of XPath supported and a review of SQL 92 syntax for predicates. It is assumed here that the reader has a working knowledge of JMS.

Current pub/sub systems are group-based. In these systems, messages are classified as belonging to a certain group, referred to as a 'Topic'. Publishers are required to label each message with a topic name, while consumers subscribe to all messages on a particular topic. For example, a topic based pub/sub system for stock trading can define a message header field in the form of a name-value pair. Publishers post messages after labeling them, by setting a header property for each particular message. Subscribers can set preferences based on predetermined header properties, known as message selectors.

The default mechanism for routing messages in JMS is the use of Message Selectors. Message selectors are created in the form of header-properties of messages (for both PTP and Pub/Sub messaging). Client applications use an SQL-92 syntax language to specify the messages to be selected.

This approach has the following drawbacks:

- Requires high processing overheads on the publisher when setting message header properties appropriately.
- Limits the scope of messages to a domain specific set because the selectors for the subscribers have to be in accordance with the message header properties that areset. Thus, there is loss of flexibility in the selection process when it comes to changing domains.

The newer generations of pub/sub systems offer a better alternative to message-selection, known as content-based routing. These systems route messages to subscribers based on content instead of message-header properties. There is no overhead imposed on the publisher and prior knowledge of the domain is not required. Subscribers have the added flexibility of choosing filtering criteria along multiple dimensions, without requiring that groups be pre-defined.

In a trading example for group based systems, the subscribers can only select trades by issue name. In contrast, the content-based subscriber is free to use an orthogonal criterion, such as volume; or a collection of criteria, such as issue, price or volume. Management of content-based systems is simpler, as there is no administration overhead required to pre-define and maintain groups. Content-based routing systems enable efficient and scalable message distribution.

Going back to the stock trading example, consider a brokerage firm that can have thousands of subscribers interested in information on stock trade. Each subscriber has their own selection criteria based on individual requirements. In the event that a subscriber would like to be alerted when two stocks fall below a certain price:

```
MSFT stock falls below 55 AND ORCL stock falls below 15
```

In another event, a subscriber would like to be alerted when any one of three stocks change price and when the market volume exceeds a certain threshold:

```
(INTC < 21 OR CSCO > 20 OR GE > 32) AND DowVolume > 1,000,000,000
```

In an unrelated event, a subscriber would like to be alerted when a Formula 1 driver sets a new lap record at Imola:

```
(Car_Make = Ferrari) AND (Driver = Senna) AND (Circuit = Imola) AND (Lap Time < 1.01
mins)
```

The information a subscriber can be interested in is unlimited. Messages, events and alerts can be desired for items such as changes in inventory, new purchase orders, product delivery, receipt of a ‘request-for-a-quote’ and late breaking news.

To efficiently implement a pub/sub system with content-based routing, an efficient solution to the problem of matching a message against a large number of subscribers, referred to as the matching problem, needs to be solved. Additionally, there must be an efficient and easy to understand standards based language that enables subscribers to register and store their personal message preferences. FioranoMQ CBR provides these functions.

Using FioranoMQ Content Based Routing

This section explains how to use FioranoMQ Content Based Routing (CBR) with samples and a description of XML syntax. FioranoMQ CBR is based on JMS 1.1 specifications with extensions to support content-based routing.

Setting up the FioranoMQ Server for CBR

The Content Based Routing support of FioranoMQ is available only in the Publish/ Subscribe JMS domain and not in the PTP (Point to Point) domain. By default, the CBR support is not enabled on the server. It can be enabled using the Fiorano Studio. Perform the following steps to set up the FioranoMQ server for CBR:

1. Start the Fiorano Studio. Select **Tools** > **Configure Profile** from the menu bar, select the profile directory in the resulting. Select Profile Directory dialog box and click the **Open** button. This switches the FioranoMQ environment to offline mode.
2. Navigate to **FioranoMQ** > **Fiorano**> **etc**> **FMQConfigLoader** in the Profile Explorer pane. The properties are displayed in the Properties pane.
3. Select **UseFioranoCbr** property and set its value to **True** from the drop-down list.
4. Right-click the root node in the profile explorer and select the **Save** option from the shortcut menu.

At the end of these steps CBR is enabled in the server and clients can send and receive XML messages with XPath selectors. More information on client side changes is available in subsequent sections.

FioranoMQ CBR XPath Support

FioranoMQ CBR utilizes a subset of XPath notation and SQL92 syntax to specify XPath message selectors.

Only absolute paths can be used. It is possible to combine several XPath string with AND/OR conditions. For example, to provide an XML:

```
<Sports>

      <Soccer>

            <Team>Manchester United</Team>

       <Record>33,5,1</Record>

       <LastGame>

                   <Team>Arsenal</Team>

                   <Date>4/5/01</Date>

                   <HomeAway>Home</HomeAway>

                   <HomeScore>2</HomeScore>

                   <VisitorScore>0<VisitorScore>
       </LastGame>

 </Soccer>

</Sports>
```

In the above example, if only messages for Manchester United are to be viewed, then the following XPath message selector can be used:

```
"/Sports/Soccer/Team = 'Manchester United'"
```

In the above example, if only messages where Manchester United was the home team are to be viewed, the following XPath message selector can be used:

```
"/Sports/Soccer/Team = 'Manchester United'

and /Sports/Soccer/LastGame/HomeAway = 'Home'"
```

Publishing XML Messages

For the Publisher sample to use CBR, set the value of InitialContext environment variable UseFioranoCbr to `true`. To use CBR related proprietary APIs, the package `fiorano.jms.runtime.xpubsub` needs to be imported. XML messages are sent as contents of FioranoXMLMessages.

FioranoXMLMessages in JMS terminology are text messages and can be created using FioranoTopicSession.

Below is a code snippet for creating a publisher with which to publish XML messages.

```
 /**
 * @(#)FCRPublisher.java           1.0, 04/25/2001
 *
 * Publishes 3 xml messages on the topic specified by client.
 * The mode of delivery can be both persistent and non-persistent.
 * One of these messages would be received by the XCRSubscriber sample.
 *
 * Copyright (c) 2001 by Fiorano Software, Inc.,
 * Los Gatos, California, 95030, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Fiorano Software, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * enclosed with this product or entered into with Fiorano.
 */ [Style check please]
import javax.jms.*;
import java.util.*;
import javax.naming.*;
import fiorano.jms.services.msg.def.*;
import fiorano.jms.runtime.xpubsub.*;
import fiorano.jms.runtime.naming.FioranoJNDIContext;
```

```
public class FCRPublisher
{
    public static void main( String args[] ) throws Exception
    {
        Hashtable env = new Hashtable();
        env.put (Context.SECURITY_PRINCIPAL, "anonymous");
        env.put (Context.SECURITY_CREDENTIALS, "anonymous");
        env.put (Context.PROVIDER_URL,
                  "http://localhost:1856"  );
        env.put (Context.INITIAL_CONTEXT_FACTORY,
                    "fiorano.jms.runtime.naming.FioranoInitialContextFactory");

        InitialContext ic = new InitialContext (env);
        TopicConnectionFactory tcf =
                        (TopicConnectionFactory) ic.lookup ("primaryTCF");

        TopicConnection topicConnection = tcf.createTopicConnection();

        Topic topic = (Topic)ic.lookup("primaryTopic");
        FioranoTopicSession topicSession =
(FioranoTopicSession)topicConnection.createTopicSession(false,1);
        TopicPublisher topicPublisher = topicSession.createPublisher(topic);

        System.out.println("Starting message delivery ....");

        FioranoXMLMessage tMsg1 = topicSession.createXMLMessage();
        FioranoXMLMessage tMsg2 = topicSession.createXMLMessage();
        FioranoXMLMessage tMsg3 = topicSession.createXMLMessage();


        String[] sym = {"MSFT","IBM","HWP"};
        int[] price = {40,50,60};
        String[] xml = new String[3];

        for(int i=0;i<3;i++)
        {
            xml[i] = "<quote>";
            xml[i] = xml[i] +  "<symbol>"+ sym[i] +"</symbol>";
            xml[i] = xml[i] +  "<askprice>"+ price[i] +"</askprice>";
            xml[i] = xml[i] +  "</quote>";
        }
        try
        {
            // setting the xml to the text messages
            tMsg1.setText(xml[0]);
            tMsg2.setText(xml[1]);
            tMsg3.setText(xml[2]);

            tMsg1.setJMSDeliveryMode (DeliveryMode.NON_PERSISTENT);
            tMsg2.setJMSDeliveryMode (DeliveryMode.NON_PERSISTENT);
            tMsg3.setJMSDeliveryMode (DeliveryMode.NON_PERSISTENT);

            topicPublisher.publish(tMsg1);
            System.out.println("Published the message :: " + xml[0]);
            topicPublisher.publish(tMsg2);
            System.out.println("Published the message :: " + xml[1]);
            topicPublisher.publish(tMsg3);
            System.out.println("Published the message :: " + xml[2]);
        }
        catch( Exception e )
        {
            System.out.println("Exception in publishing:" +e );
        }
    }
}
```

Subscribing to XML Messages

Creating a FioranoMQ CBR subscriber is similar to creating a JMS subscriber. Set the value of `InitialContext` environment variable `UseFioranoCbr` to `true`. The subscribers (durable and non-durable) can then be created in a manner similar to PubSub. The message can be any valid XML message passed as the contents of FioranoXMLMessages.

CreateDurableSubscriber

The following API creates a Durable Subscriber which receives messages only if they conform to the specified Xpath Message Selector string:

```
 /**
 * Fiorano's proprietary API to create XPath Durable Subscriber.
 *
 * This API creates an XPath Durable Subscriber.
 *
 * @param topic - the non temporary topic to subscribed to.
 * @param subscriptionID - ID used to identify subscription.
 * @param messageSelector - string containing the XPth message selector or normal JMS
Message selector.
 * @param NoLocal - if set, inhibits delivery of messages published over its own
connection.
 * @exception JMSException if operation fails
 *
 */
public TopicSubscriber createDurableSubscriber (

                                Topic topic,
                                String subscriberID,
                                String messageSelector,
                                boolean noLocal)
                       throws JMSException ;
```

CreateSubscriber

The following API creates a Non-Durable Subscriber, which only receives messages if they pass the specified Xpath Message Selector string:

```
    /**
     * Fiorano's proprietary API to create XPath Subscriber
     *
     * Create XPath Durable Subscriber
     *
     * @param topic - the non temporary topic to subscribe to.
     * @param messageSelector - string containing the XPth message selector or normal
JMS Message selector.
     * @param NoLocal - if set, inhibits delivery of messages published by its own
connection.
     * @exception JMSException if operation fails
     */
     public TopicSubscriber createSubscriber (

                           Topic topic,
                           String messageSelector,
                           boolean noLocal)
                 throws JMSException  ;
```

To create a FioranoMQ CBR subscriber, you can follow the example code snippet given below, which sets up an XPath Subscriber, onMessage and onException listeners.

```
/*
 * Copyright (c) 2001, Fiorano Software, Inc.
 * All Rights Reserved
 *
 * FileName   : XPathSubscriber.java
 *
 * [No Text. Check please]
 * Implements an asynchronous listener to listen
 * for messages published on the topic - "primaryTopic"
 * only receiving message which match the selector parameter
 *
 * Questions/comments/suggestions?
 * Please visit: http://www.fiorano.com
 * Or e-mail: support@fiorano.com
 *
 * @since     FioranoMQ 6.0, August 2002
 */

import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
import fiorano.jms.services.msg.def.*;
import fiorano.jms.rtl.*;
import fiorano.jms.runtime.naming.FioranoJNDIContext;

import java.net.*;

class Subscriber implements MessageListener,ExceptionListener
{
    public static void main (String args[])
    {
       Subscriber subscriber = new Subscriber ();
        try
        {

            // 1. Create the InitialContext Object used for looking up
            //     JMS administered objects on the Fiorano/EMS
            //     located on the default host.
            //
            Hashtable env = new Hashtable ();
            env.put (Context.SECURITY_PRINCIPAL, "anonymous");
            env.put (Context.SECURITY_CREDENTIALS, "anonymous");
            env.put (Context.PROVIDER_URL,
                     "http://localhost:1856");
            env.put (Context.INITIAL_CONTEXT_FACTORY,
                     "fiorano.jms.runtime.naming.FioranoInitialContextFactory");
           InitialContext ic = new InitialContext (env);
            System.out.println ("Created InitialContext :: " + ic);

            // 1.1 Lookup Connection Factory and Topic names
            //
            TopicConnectionFactory tcf =
                        (TopicConnectionFactory) ic.lookup ("primaryTCF");
            Topic topic = (Topic)ic.lookup("primaryTopic");


            // 2. create and start a topic connection
            System.out.println("Creating topic connection");
            TopicConnection topicConnection = tcf.createTopicConnection();
        // Register an Exception Listner
        topicConnection.setExceptionListener (subscriber);
            topicConnection.start ();

            // 3. create topic session on the connection just created
            System.out.println("Creating topic session: not transacted, auto ack");
            TopicSession topicSession = topicConnection.createTopicSession(false,1);

            // 4. create XpathSubscriber
            System.out.println("Creating topic, subscriber");
            String selector = "/quote/symbol = 'IBM' and /quote/askprice > 40";
            TopicSubscriber topicSubscriber =
              topicSession.createSubscriber(topic,selector,false);

            // 5. install an asynchronous listener/callback on the subscriber object
            //    just created
            System.out.println ("Ready to subscribe for messages :");
            topicSubscriber.setMessageListener (new Subscriber ());
        }
        catch (Exception e)
        {
            e.printStackTrace ();
        }
    }

    /**
     * Message listener which receives messages aysynchronously
     * For the bound subscriber.
     */
    public void onMessage( Message m )
    {
        if( !(m instanceof FioranoXMLMessage) )
        {
            return;
        }

        String s = "";
        try
        {
            s = ((FioranoXMLMessage)m).getText();
        }
        catch( Exception e )
        {
            System.out.println("Exception in getText() :" + e );
        }
        System.out.println ("Received the message :: "+s);
    }
    /**
     * If a JMS provider detects a serious problem with a
     * Connection this method is invoked passing it a JMSException
     * describing the problem.
     * @param JMSException e
     */
    public void onException (JMSException e)
    {
//Report the Error and take necessary Error handling measures
        String error = e.getErrorCode ();
System.out.println (error);

        ((fiorano.jms.common.FioranoException)e).printCompleteStackTrace();
    }
}
```

More samples of using FioranoXCR content-based routing are available in the samples directory of the FioranoMQ installation under pubsub/ContentBasedRouting.

Handling Massive Number of Subscribers

In an environment where millions of subscribers are required, it is possible to cascade FioranoMQ Servers a form that allows data source publishers to multiple FioranoMQ servers simultaneously. This enables each source to publish and/or send messages to several FioranoMQ Servers. Each of these in turn sends the information to another set of FioranoMQ Servers or sends information directly to the system that ultimately transfers information to subscribers. For best results it is recommended not to run more than 500 subscribers on each FioranoMQ connection.

Below are the key concepts that enable the handling of a large number of subscribers:

- Keep XML messages small (do not mix domain data, such as weather and sports. Do not add information on more than one item within a domain at a time, such as stock quotes for more than 1 company).

- Use fewer connections and more subscribers per connection.
- Use suitably powered systems that run subscribers so that they do not decelerate the movement of messages. Most slowdowns occur due to subscribers that are unable to process the messages at speed.
- Segregate domain data on different topics (for example, use a topic for weather and a different topic for sports).

Following figure illustrates an environment where incoming XML messages are distributed to several FioranoMQ boxes. These FioranoMQ boxes in turn feed the distribution systems that eventually send message to devices such as cell phones and wireless PDAs. This example illustrates how multiple subscribers can be used to distribute data on several different topics.

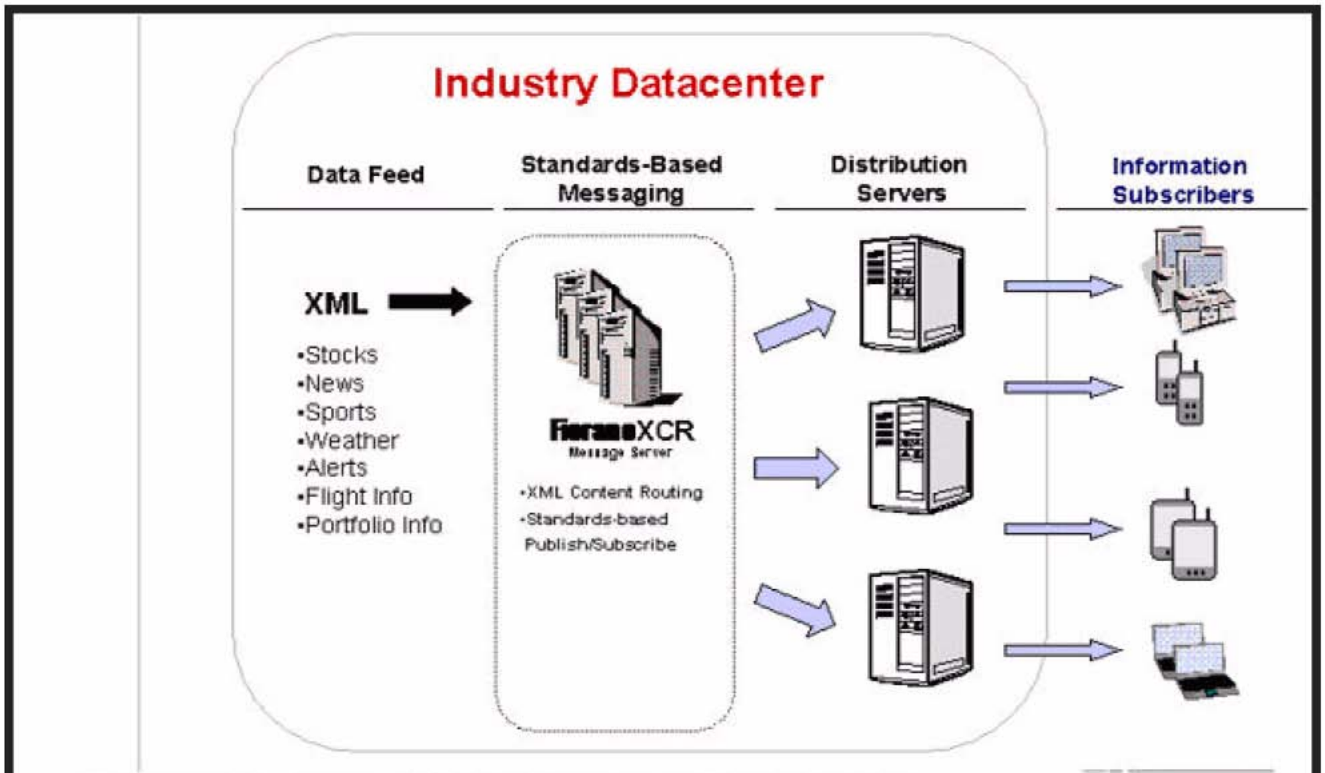


FIGURE 4-1 Handling Large Number Of Subscribers

XML Support

The FioranoMQ Content Based Routing supports elements and attributes typical to XML. Later versions of FioranoMQ support all XML files. It is recommended that XML files remain small so that they can be parsed and delivered quickly. Large XML files take longer to parse and distribute. While formatting XML, several methods are available to package the information. This implies that information can be packaged in Elements only, in Attributes only, or in Elements and Attributes xml. Depending on the packaging, it can be optimized for CBR.

For example, if a subscriber wants to receive stock information from company XYZ whereas the XML file contains stock information on companies XYZ and company ABC, then the subscriber receives some information it does not require or need. Therefore, while disseminating stock quotes, it is preferable to add information pertaining to only one stock in a single message.

This ensures that:

- Messages remain small.
- Unwanted content is not sent to the subscribers.

For the purpose of discussion XML files can be broken into three categories: Elements only, Attributes only, and Elements and Attributes. Examples of each type of XML, as well as example message selectors can be found in the section below. When wrapping data in XML, so as to be optimized for content-based routing, it is recommended that 'elements only' and/or 'attributes only' be used to enable highest message throughput.

An example of elements only is:

```
<atag>
 <btag>value</btag>
 <ctag>value</ctag>
     .
     .
     .
<ztag>value</ztag>
</atag>
```

In the above example, the level of nesting is not fixed to two, but shows that this XML type contains only elements. This type of XML is the fastest to process.

An example of attributes only is:

```
<Stock quotes>
 <Quote Symbol="x" ASkPrice="y" BidPrice="z">
 <Quote Symbol="a" AskPrice="b" BidPrice="c">
</Stock quotes>
```

The above example shows that there are only attributes with no elements that can be referenced. This does not imply that an elements only XPATH, such as "/Stock quotes/Quote Symbol IS NOT NULL," cannot be used. This selector indicates that any XML message containing this element can be selected. This type of XML is a little slower to process than 'elements only' type of XML.

An example of elements and attributes is:

```
<atag>
 <btag att1="x" att2="y">
   <ctag att3="z">value</ctag>
 </btag>
</atag>
```

The above example shows how elements can be found within the elements containing attributes. Processing this type of XML can be much slower than processing either of the other two types of XML, depending upon the size of the message. This XML would require more complex XPath identifiers.

FioranoMQ Content - Based Message Selector Language

This section explains SQL92 syntax predicates and the subset of XPath supported by FioranoMQ CBR. In addition, it explains the type of XMLs that are supported, along with suitable examples of the XMLs and message selectors.

General Form

Message selection criterion is registered by consumers using the FioranoMQ Content-based Message Selector (CbMS) language, which is a subset of the SQL92 conditional expression syntax. It is combined with XPath like notation for retrieving identifiers and values from XML documents.

The order of evaluation of a CbMS takes place from left to right incorporating within it levels of precedence. Parentheses can be used to change this order. Predefined selector literals and operator names are written in uppercase though they are case insensitive.

XPath notation is used as a reference mechanism to return one or many elements from an XML file. The entire XPath specification is sometimes inaccurate for content-based message selection, particularly where speed and scalability is essential. XPath notation must return only a single value, else an exception is launched.

The general form of CbMS is:

```
[/ {not} C {[in]|[between]} q y {and|or} /]
```

Where:

"[/ /] One or many.

"{} Optional.

"{|} Optional, and if used then select one C Identifier: For details refer to the section on 'Identifiers'.

"q Operator. For details refers to the section on 'Operators'.

"y Literal. For details refer to the section on 'Literals'.

Examples:

Below are a few examples of the FioranoMQ content-based message selector language:

```
/quotes/symbol = 'IBM' (NOTE: C is '/quotes/symbol', q is '=' and y is 'IBM')
/quotes/symbol='IBM' or /quotes/symbol='CSCO'
(/quotes/symbol='IBM' or /quotes/symbol='CSCO') and /quotes/bidprice>150
/quotes/symbol in ['IBM','CSCO'] and (/quotes/pricedelta between 1 and 10)
```

Subset of Supported XPath Queries

The current limitations of CbMS XPath as compared to the full XPath specifications are:

1. **Only absolute paths are allowed**.

   This indicates that all the XPath strings must start with '/'. '//' is not supported for relative paths. All the paths must be specified from the root of the XML. For example, the following XPath query is invalid:

   ```
   "//person"
   ```

   If a person is found under parent researchers, then the correct XPath query would be: [

   ```
   "/researchers/person"
   ```

2. **A test node can have only one predicate**.

   This indicates that the query below is invalid, even though it can be valid in XPath:

```
"//researchers/person[@name = "Shin"][@loc = "Bethesda"]"
```

   The query below is acceptable:

```
"/researchers/person[@name = "Shin"] and /researchers/person[@loc = "Bethesda"]"
```

3. **A predicate cannot appear inside another predicate**.

   This indicates that the query below is invalid since it has a predicate nested inside another predicate.

```
"/researchers/person[@name = /salesperson/person/name[@id > "8080"]])"
```

4. **The character '|' and the expressions 'and' and 'or' are not supported inside a predicate**.

   This implies that the operator "|" (Union operator) is not allowed for connecting path expressions outside predicates. Connecting path expressions outside predicates is accomplished through the use of the 'OR' keyword. Inside predicates, "|", "and" and "or" operators are not allowed. Therefore, the following query is invalid:

```
"/a[@id="1" or in("2","3")]"
```

   The above invalid query becomes a valid query when represented as:

```
"/a[@id="1"] or /a[@id=2] or /a[@id=3]"
```

5. **One side of the equality and relation operator (=, <, >, <=, >=) must be a literal**.
6. **The 'join' operation is not supported**.

   Points (5) and (6) indicate that comparison inside predicates is limited. At least one argument must be literal for the equality and comparison operator. For instance, the query below is valid:

```
"//a[@id > "100"]" or "//a[title = "XPERT"]"
```

   Whereas, the query below is invalid:

```
"//a[@id=//b/@id]"
```

   At present, the 'semi-join' operation is not supported.

7. **Only three functions, "contains()", "in()", and "between()" are supported**.

This indicates that only three functions are supported; "contains()", "in()", and "between()".

8. **"*" representing 'anything' is not supported**.

   This indicates that the wildcard character "*" is not supported in any form.

Identifiers

Identifiers are expressed in terms of XPath notation. XPath can be used to refer to any part or parts of an XML document. Since message selection is based on actual contents of a message and not just upon the presence of the message, the XPath notation must return a single attribute of an element in the XML file. Any valid XPath notation is supported, but in case an XPath returns multiple elements, only the first one will be evaluated. The following XPath notation is recommended:

"/ROOT/CHILD where CHILD contains only 1 element

"/ROOT/CHILD[x] where x is the number of exact element of type CHILD

"/ROOT/CHILD[last()] selects the last element of type CHILD

"/ROOT/CHILD[first()] selects the first element of type CHILD

Operators

XPATH selectors involve the use of various arithmetical, logical and conditional operators. FioranoMQ CBR supports most of the operators. This section explains, with examples, the support of the following operators:

Standard bracketing () is supported. This implies that the conditions inside a bracket are evaluated first followed by the conditions outside the bracket.

Logical operators in order of precedence: NOT, AND, OR

Comparison operators: =, >, >=, <, <=, <>

Note - Not equal (<>) is supported internally as (NOT (C = y))

Only like type values can be compared. A string can be compared to a string and a boolean to a boolean. However, it is valid to compare exact numeric values and approximate numeric values (the type of conversion required is defined by the rules of Java numeric promotion). If the comparison of non-like type values is attempted, then the selector is always false.

String and Boolean comparison is restricted to = and <>. Two strings are equal if and only if they contain the same sequence of characters.

Note - Not equal (<>) is supported internally as (NOT (C = y))

"-+, - unary

"*, / multiplication and division

"+, - addition and subtraction

"Arithmetic operations must use Java numeric promotion

"{NOT} BETWEEN arithmetic-expr1 and arithmetic-expr2 (exact numeric values only)

Note - BETWEEN x and y is supported internally as z >= x AND z <= y

NOT BETWEEN x and y is supported internally as z < x OR z > y

" {NOT} IN (string-literal1, string-literal2...)

Note - IN (x,y,z) is supported internally as w = x OR w = y OR w = z

Note - NOT IN (x,y,z) is supported internally as ((NOT(w = x)) AND (NOT(w = y)) AND (NOT(w = z))))

(OPTIONAL) {NOT} LIKE pattern-value [ESCAPE escape-character] comparison operator, where identifier has a String value. Pattern-value is a string literal, where '_' represents any single character; '%' represents any sequence of characters (including the empty sequence); and all other characters represent themselves. The optional escape-character is a single character string literal, whose character is used to release the special meaning of '_' and '%' in pattern-value.

Below are few examples of the use of the LIKE operator with the results:

- "Phone LIKE '12%3' is true for '123' '12993' and false for '1234'
- "Word LIKE 'l_se' is true for 'lose' and false for 'loose'
- "Underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'
- "Phone NOT LIKE '12%3' is false for '123' and '12993' and true for '1234'
- "If identifier of a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.
- "{NOT} NULL allows for testing of the existence of an element or elements within an XML without actually testing any values.

Examples:

/quotes/bidprice NOT NULL succeeds for any XML which has any elements matching /quotes/bidprice /quotes/bidprice. NULL succeeds for any XML which has no elements matching /quotes/bidprice

Literals

A string literal is enclosed in single quotes with a single quote included, represented by doubled single quote such as 'literal' and 'literal''s'. Similar to Java String literals, these use the Unicode character encoding.

An exact numeric literal is a numeric value without a decimal point: 57, -957, +62. Numbers in represented within the range of ‘Java long’ are supported. Exact numeric literals use the Java integer literal syntax.

An approximate numeric literal is a numeric value in scientific notation: 7E3, -57.9E2 or a numeric value with a decimal such as 7., -95.7, +6.2. Numbers represented within the range of 'Java double' are supported. Approximate literals use the Java floating point literal syntax.

The two Boolean literals are 'TRUE' and 'FALSE'.

Example XMLs

The complete XPath specification can be found at: http://www.w3.org/TR/xpath

Below are three examples of XML files and associated XPath strings that are found within the XML files. For each XML file there is an example message selector. After each sample message selector are the strings that need to be returned by the single pass parser. Following each string is the value that is returned by the single pass parser, given the value of the particular XPath string is found to be advantageous. If the value of the XPATH string is found matching in the parsed XML, then the message is selected.

Elements Only XML

```
<Stock quotes>
 <Symbol>csco</Symbol>
 <AskPrice>33.50></AskPrice>
 <BidPrice>33.25>>/BidPrice>
</Stock quotes>
```

Example 1 for Elements Only XML

An example of a related XPath predicate:

```
/Stock quotes/AskPrice > 33.2 and /Stock quotes/Symbol = 'csco'
```

Description: Generate the XML if AskPrice is greater than 33.2 and Symbol is 'csco'. This is an example of using equality and inequality check."

Example 2 for Elements Only XML

An example of a related XPath predicate:

```
/Stock quotes/AskPrice > 33.2 and /Stock quotes/Symbol LIKE 'csc%'
```

Description: Generate the XML if AskPrice is greater than 33.2 and Symbol starts with 'csc'.

An example of using Ineq and LIKE trees:

XPath parser will return:

```
/Stock quotes/Symbol
/Stock quotes/AskPrice
/Stock quotes/BidPrice
```

Values returned for each XPath string:

```
/Stock quotes/Symbol 'csco'
/Stock quotes/AskPrice '33.50'
/Stock quotes/BidPrice '33.25'
```

Attributes Only

```
<Stock quotes>
 <Quote Symbol>="csco" AskPrice="33.50" BidPrice=33.25">
 <Quote Symbol>="ibm" AskPrice="122.50" BidPrice="122.25">
<Stock quotes>
```

An example of a related XPath predicate:

```
/Stock quotes/Quote[@Symbol='csco'] NOT NULL and /Stock quotes/Quote[@AskPrice] > 33.2
```

Description: Generate the XML if it contains a Quote Element with attribute Symbol set to 'csco' and an AskPrice attribute > 33.2. An example of using Ineq and NULL trees is given below. This example demonstrates the use of checking attribute values inside the XPath query.

XPath parser will return:

```
/Stock quotes/Quote[@Symbol]
/Stock quotes/Quote[@AskPrice]
/Stock quotes/Quote[@BidPrice]
/Stock quotes/Quote[@Symbol]
/Stock quotes/Quote[@AskPrice]
/Stock quotes/Quote[@BidPrice]
```

Values returned for each XPath string:

```
/Stock quotes/Quote[@Symbol]
'csco
/Stock quotes/Quote[@AskPrice]
'33.50'
/Stock quotes/Quote[@BidPrice]
'33.25'
/Stock quotes/Quote[@Symbol]
'ibm'
/Stock quotes/Quote[@AskPrice]
'122.50'
/Stock quotes/Quote[@BidPrice]
'122.25'
```

Elements and Attributes XML

```
<Quotes>
 <QuoteType Type='Quote of the Day'>
```

'This is the funny quote of the day for April 2, 2001 [Close quotation? Syntax check please]

```
 </QuoteType>
 <QuoteType Type='Stock Quote'>
  <Symbol>csco</Symbol>
  <AskPrice>33.50</AskPrice>
  <BidPrice>33.25</BidPrice>
</QuoteType>
```

An example of a related XPath predicate:

```
/Quotes/QuoteType[@Type='StockQuote']/Symbol = 'csco' and
/Quotes/QuoteType[@Type='StockQuote']/AskPrice > 33.2
```

Description: Generate the XML if it contains QuoteType element with attribute Type = 'StockQuote' and elements = 'csco' and > 33.2. The following example uses Eq and Ineq trees:

XPath parser will return:

```
/Quotes/QuoteType[@Type='Quote of the Day']
/Quotes/QuoteType[@Type='Stock Quote']/Symbol
/Quotes/QuoteType[@Type='Stock Quote']/AskPrice
/Quotes/QuoteType[@Type='Stock Quote']/BidPrice
```

Values returned for each XPath String are given below:

```
/Quotes/QuoteType[@Type='Quote of the Day']
         'This is the funny quote of the day for April 2, 2001'
/Quotes/QuoteType[@Type='Stock Quote']/Symbol
         'csco'
/Quotes/QuoteType[@Type='Stock Quote']/AskPrice
         '33.50'
/Quotes/QuoteType[@Type='Stock Quote']/BidPrice
         '33.25'
```

Limitations of FioranoMQ Content Based Routing

Given below are a few of the known limitations of FioranoMQ Content Based Routing:

1. If an attribute or a tag does not exist and a message selector attempts to check the presence of that attribute or tag, then the message is not selected, irrespective of the nature of the selector. For instance, consider the following XML as an example:

   ```
   <Stock quotes>
   <Symbol>csco</Symbol>
   <AskPrice>33.50</AskPrice>
   <BidPrice></BidPrice>
   </Stock quotes>
   ```

   A message selector in the following form will return a 'false':

   ```
   //Stock quotes/Symbol/AskPrice[@att1='0']=33.50
   //Stock quotes/Symbol/AskPrice[@att1<>'0']=33.50
   ```

   This is because no attribute by the name 'att1' exists in the XML message and the message is not delivered.

   Similarly, a selector in the following form will return a 'false' and the message will not be delivered:

   ```
   //Stock quotes/Symbol/BidPrice<>212
   ```

2. 'Between' is not supported in attributes. It is, however, supported tags.

   Consider the XML given below:

   ```
   <Stock quotes>
   <Symbol>csco</Symbol>
   <AskPrice att1='12'>33.50</AskPrice>
   <BidPrice att2='13'>33.25</BidPrice>
   </Stock quotes>
   ```

   A selector in the following form is not supported;

   ```
    /Stock quotes/Symbol/AskPrice[@att1 between '12' and '13']:
   ```

   However, a selector in the following form is supported:

   ```
   /Stock quotes/Symbol/AskPrice between 30 and 40
   ```

3. Message selectors involving mathematical operations are not supported. For example, consider the XML given below:

   ```
   <Stock quotes>
   <Symbol>csco</Symbol>
   <AskPrice>33.50</AskPrice>
   <BidPrice>33.25</BidPrice>
   </Stock quotes>
   ```

   A message selector in the following form is not supported:

   ```
   /Stock quotes/Symbol/AskPrice = 0.3350*100
   ```

4. JMS message selectors in the following 'type' are not allowed:

Consider a TextMessage on which the following properties have been set:

```
textMessage.setIntProperty("low_val",10);
textMessage.setIntProperty("high_val",100);
textMessage.setIntProperty("mid_val",50);
```

A message selector in the following form is not supported:

```
mid_val between low_val and high_val
```