OSMQ™

Open Source Message Queue

Developer's Guide

MQue Systems™

OSMQ™ Developer's Guide

MQue Systems™

Systems that Plug-and-Share™

Introduction

MQue Systems

MQue Systems is a company with a single focus -- integration technology. We are committed to providing our customers with an open architected integration model and suite of tools that are the most reliable, easily maintainable, and highest performing in the industry.

OSMQ

MQue's is a third generation message middleware suite that delivers unprecedented performance and reliability. This 100% Java™ integration broker runs on any Java 2 SE compliant platform, including Sun Solaris™, Microsoft Windows™, and Linux.

OSMQ was designed to meet the most demanding capacity and reliability requirements. It is uniquely capable of integrating high-volume, mission critical financial systems, and its flexibility and extensibility makes it the ideal choice for any EAI or business-to-business distributed integration task.

Publish Everything

OSMQ is the basis of MQue Systems' InteBroker™ development suite. InteBroker is a GUI version of the OSMQ message broker that includes a suite of adapters which provide outstanding support for MQue's publish everything™ distributed architectural model. Publish everything is part of the MQue Systems EIM design model that supports high-volume financial data movement, enabling MQue Systems customers to maintain databases throughout a geographically distributed business organization that constantly represent the current state of a corporate or business-to-business enterprise.

Whether you need to distribute and integrate all the market activity of a stock exchange, rapidly and reliably collect and publish telecom switch information from locations that span several continents, or ensure that several departmental database servers always reflect the information being maintained by the many corporate legacy systems, InteBroker can provide the reliability, performance and overall simplicity that you have not been able to achieve.

Messages

Note - This guide is intended primarily for developers who design and build record and transaction oriented business applications. Although OSMQ supports the distribution of messages that contain unformatted binary data, the message format described in this document will be primary focused on *data set* formatted messages.

Defining Messages

One of the most important tasks of an integration project is defining an appropriate set of messages. A message is the unit of information that is used to communicate the details of enterprise-level *commitment events* and *service requests*. (For more information regarding commitment events and service requests, refer to the Enterprise Integration Model documentation.)

MQue recommends that you adopt an *entity relational* message model. An entity relational message model defines a message as a relational data entity that contains data attributes. A major benefit of the entity relational model is the ability to map messages into relational database tables.

The process of defining messages is similar to defining database tables. It involves traditional relational data modeling -- assembling a collection of data entities, and normalizing those entities into various topical messages.

Topics

Each message definition is associated with a *topic*. The topic identifies the entity that is associated with the message. Topics differentiate the various *message types*. Each topic is associated with a fixed set of data attributes (message elements), and those elements have a fixed ordinal position.

For example, NorthStar Mutual Fund Company defines 3 primary message types: ***CUSTOMER***, ***ACCOUNT*** and ***FUND***. The data normalization process resulted in 50 message elements being associated with the ***CUSTOMER*** message, 200 elements associated with the ***ACCOUNT*** message, and 150 elements associated with the ***FUND*** message.

A message contains a *header* and *body* section. The header contains identifying and routing information, such as the *topic* and *the originator*. The body contains the data elements that correspond with the topic.

Physical Layout

DataSet messages are relatively self-defining. For reasons of efficiency (message size and parsing efficiency), message elements are represented internally as an ASCII delimited string of elements. The first character (byte) of the header is considered the delimiter.

The first ten elements constitute the message header, and the remaining elements are members of the message body. The header is always plain text (unencrypted). The body may or may not be encrypted. Any encryption and decryption is performed by the client adapters.

The Message Header

The message header has ten elements. Four of the header elements are mandatory: the *topic*, *type*, *originator*, and *format*.

ELEMENT	POSITION	DESCRIPTION	VALUES
RECIPIENT	1	If present, identifies the sole agent that will receive the message. This value must be set by the application.	Alpha-numeric string
TOPIC *	2	Identifies the message elements based on a business-related category. This value must be set by the application.	Alpha-numeric string, ideally coinciding with the name of a corresponding database table
TYPE *	3	Event that precipitated generation of the message. The default value is NOTIFICATION.	NOTIFICATION = N SERVICE REQUEST = R BATCH CONTROL = B TOPIC CHG REQUEST = T DENIAL OF REQUEST = D
ORIGINATOR *	4	Identifies the agent that created the message. This value is set by the PublisherBean if there is none already set by the application.	Alpha-numeric string
CONTENT FORMAT *	5	Format of the data found in the message body. This value is defaulted by the message class when the message is instantiated.	TEXT = T DATASET = D BINARY = B
DATA KEY	6	Unique (primary) key value that distinguishes the values in the message body. This value must be set by the application.	Alpha-numeric string
TRANSACTION EVENT	7	Extended information regarding type of event that precipitated the message generation. Values are those commonly used by client adapters, including database adapters. This value must be set by the application.	ADD EVENT = A DELETE EVENT = D UPDATE EVENT = U
CORRELATION ID	8	Unit of work identifier. Messages from the same originator and with the same identifier are considered part of the same unit of work. If auto commit is turned off, this value is incremented by the PublisherBean whenever there is a commit.	Integer values from 1 - n
SEND TIMESTAMP	9	Date and time when the message was created by the originator in CCYYMMDDHHMMSS format. This value is set by the PublisherBean.	19980409162306
SENDER SEQUENCE	10	An originator-specific number that distinguishes various messages generated by that originator. This value is normally set by the PublisherBean.	Integer values from 1 - n

(* Mandatory elements)

In the following example, the topic is "CUSTOMER" and the originator is "CUSTMAINT." The message type is NOTIFICATION, the activity is an ADD_EVENT, and the content format is a DATASET. There is no value for the RECIPIENT or CORRELATION ID.

```
||CUSTOMER|N|CUSTMAINT|D|982038|A||0924893|43542|
```

Message Type

The message type is the category of event that caused a message to be generated. Standard type include .NOTIFICATION ('N') and REQUEST ('R'), DENIAL ('D'), TOPIC_CHG ('T') and BATCH_CONTROL ('B'). These are defined in the osmq.messages.MessageType interface.

NOTIFICATION messages are published by an OLTP system after it commits a transaction. (Batch applications can also publisher NOTIFICATION messages.)

REQUEST messages are point-to-point routed client requests for service. (Point-to-point messages are sent to a single recipient, and have the recipient identified in the message header, versus publish-subscribe messages that have no recipient and are published to all subscribers to the message topic.

DENIAL messages are forwarded to a service requestor by a service provider. They indicate a request for service has been denied (based on some business rule or security violation).

TOPIC_CHG messages are used internally to identify a TOPIC CHANGE EVENT. These messages are generated by the SubscriberBean class when a subscriber session is opened, and they should not be sent explicitly by a client application.

BATCH_CONTROL message are sent immediately before and after a batch of related messages. They are used to delimit the batch, so that subscribers can determine an action to be taken (such as creating export files, or updating database tables). The terminating CONTROL message will normally contain totals that should be considered a checksum.

Message Routing

Message routing is based on the values of the *recipient* and *topic*. A topic value is mandatory, and a recipient value is optional.

Point-to-point routing

If a message has a recipient value, the message is routed only to that recipient. A message that has a has a recipient value is referred to as a *point-to-point* message, since it is generated by one process (the *originator*) and sent to only one process (the *recipient*.) Recipient-directed messages typically include REQUEST and DENIAL message types, although a NOTIFICATION message might be directed to a specific recipient in the case of a multi-node work-flow process.

Publish-subscribe routing

A message that has no recipient value is routed to all processes that subscriber to the topic value contained in the message. For that reason, a message that has no recipient is referred to as a *publish-subscribe* message. In the prior example, because the message has a topic of "CUSTOMER" and no recipient value, the integration broker would route the message to every process that was currently subscribed to the topic "CUSTOMER." Consistent with the prior example, a message with no recipient will normally have a message type of NOTIFICATION.

The Message Body

The message body is appended to the header. The value of the *format indicator* in the message header indicates whether the body content is binary, string, or data set. The acceptable values are contained in the osmq.messages.MessageFormat interface, and include BINARY ('B'), TEXT ('T') and DATASET ('D').

- BINARY: Binary information is represented as an unformatted array of bytes. Byte format is suitable for passing byte-oriented information, such as audio or video images. Note that this format is not supported unless the broker is being run in binary mode.
- TEXT: Text information is represented as an ASCII text string. Text format is suitable for passing textual information.
- DATASET: Data set information is delimited text-formatted information that represents record-oriented data. Data set format is suitable for publishing business-oriented event messages.

Data Set Elements

Data set elements (fields) in a data set message are referenced by their relative position. OSMQ adapters are normally used to map native data type values into data set elements. For example, the Java adapters map integer, long, double, String, BigDecimal, boolean and Date data types into data set message elements. Within the message body data set elements are represented as strings.

In the following example of a data-set formatted message, the data set contains three elements. Element one is the integer *982*, element two is a string *"SMITH"* and element three is the fractional number *12.5*.

```
||CUSTOMER|N|CUSTMAINT|D|982038|A||0924893|43542|982|SMITH|12.5|
```

The DataSetMessageBean Class

The ***DataSetMessageBean*** class is a Java bean that provides public methods for setting the various header elements, and setting the positional elements from native data type values.

The following image displays the ***DataSetMessageBean*** Java bean as it appears in the properties windows of Borland's JBuilder.

The Data Set Message API

The ***DataSetMessageBean*** is used by both publisher and subscriber applications. When writing a publisher application, you can instantiate one or more messages that will be associated with the various message entities being published. Instantiation a message involves defining the number of body elements and setting the topic.

In this example, a message is created to publish customer-related information that contains 100 record elements. The value of the message Topic is set to the value "CUSTOMER." Applications that subscribe to the topic "CUSTOMER" will receive copies of messages that are published by this application.

```
DataSetMessage message = new DataSetMessageBean();
message.setElementCount(100);
message.setTopic("CUSTOMER");
message.setType(MessageType.NOTIFICATION);
```

The `MessageFactory contains a public static method that creates` DataSetMessage instances, initializing the topic, setting the number of body elements, and setting the type to the default value of *MessageType.NOTIFICATION*. In this example, the message is set to 100 elements, with a topic of "CUSTOMERS.":

```
DataSetMessage message =
MessageFactory.createNotificationMessage(100, "CUSTOMERS");
```

The ***DataSetMessageBean*** provides API's that you can use to *get* and *set* the various body elements. There are corresponding set/get methods to put and get the various elements as native data type values. The syntax of these methods is consistent with the syntax of the positional methods defined in Sun's standard Java SQL package.

In this Java example, the fifth element in the message body is set to the integer value *570*, and the seventh element is set to the string value *"BOSTON."*

```
message.setInteger(5, 570);
message.setString(7, "BOSTON");
```

A corresponding subscriber can reference the message elements with a set of reciprocal `getxxx()` methods

```
int myint = message.getInteger(5);
String mystring = message.getString(7);
```

For a detailed description of the public methods supported by the ***DataSetMessageBean*** class, see the Java documentation included with the OSMQ Developers Suite.

Publisher and Subscriber Applications

OSMQ supports an open architected client model. Any application program that can create a socket connection and build / parse a delimited string of message header and body elements can function as a publisher or subscriber of text and data set messages.

Designing adapters for binary-mode messages is also possible but somewhat more complex, since they require the inclusion of a 4-byte size indicator (in binary notation)

Note - If you are developing publisher or subscriber applications and are building your own message adapter, it is recommended that you run OSMQ in non-binary mode. Binary mode messages must include a 4-byte binary value that identifies the message size.

Client Connections

Clients use standard TCP/IP socket sessions to communicate with the integration broker. A client will normally read and write text stream information.

The following information describes connection information for non-Java clients. (Java client applications using the SubscriberBean and PublisherBean classes are not required to explicitly send this information since it is performed when a session is opened).

After connecting to the integration broker, the client process writes a spaces delimited text string that identifies the client and its intended role. The first character in the string identifies the client's role as a publisher, subscriber, or topic change requestor. The role code values are 'P' for publisher, 'S' for subscriber, and 'T' for topic change requestor. The second value in the string is the client's identifier. The identifier is a name that uniquely distinguishes the client from all other processes that can connect to the server.

In this example, the client identified as 'CUSTMAINT' is connecting as a publisher.

```
P CUSTMAINT
```

Topic Change Connections

In order to receive topical messages, an application must first connect to the broker and register as a *topical subscriber*. The application must also connect to the broker and "unsubscribe" if it no longer wants topical messages to be queued.

The subscriber connects as a topical client, and then writes one or more subscription change strings. The format for each topical change request is a '+' or '-' (indicating subscribe or un-subscribe) and the name of the topic. In this example, the client is connecting as the subscriber name '*WEBSERVER*,' and is subscribing to three topics:

```
T WEBSERVER
+ CUSTOMER
+ TRADES
+ ORDERS
```

Similarly, in this example, the same application is un-subscribing to the same three topics:

```
T WEBSERVER
- CUSTOMER
- TRADES
- ORDERS
```

Message Transfer Modes

When a client connects to the message broker, it determines whether the broker was opened in text or binary message transfer mode. If the broker was opened in text mode, it indicates that a '\r' character is appended to the end of each message that it is written to the output stream. If the broker was opened in binary mode, a 4-byte integer that defines the message size is pre-pended to the message. (The result is that a binary message can be over 2 billion bytes in length.) Publisher and subscriber clients automatically handle writing and reading messages according to the prescribed mode. Note that messages with a content format of BINARY (those created with a BinaryMessageBean) can only be written and read if the broker was opened in binary message mode, whereas messages with a content format of DATASET or TEXT can be written and read in either broker mode.

Java Classes and Components

The OSMQ Developers Suite includes a comprehensive set of classes and Java bean components that can be used to build publisher and subscriber applications.

Java Bean Components and Adapters

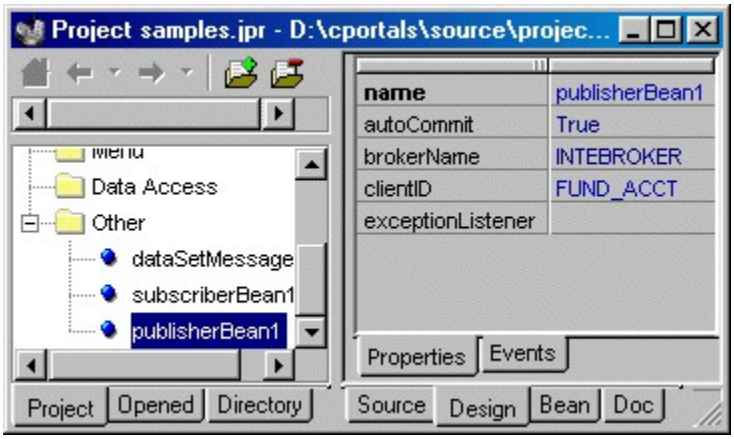
The ***PublisherBean*** and ***SubscriberBean*** components are Java bean adapters that can be used to build subscriber and publisher applications. These components simplify and automate many of the tasks related to publishing and subscribing to messages.

The adapters can be installed on the Java bean palette of most integrated development tools, such as IBM's Visual Age for Java, Borland JBuilder, and Sun's Forte for Java. See your IDE documentation for details regarding installing Java bean components.

The PublisherBean Adapter

The PublisherBean adapter is used to publish topical and point-to-point messages. The PublisherBean is the preferred application-level component for publishing messages. Several of the more frequently referenced publisher attributes can be set directly within the bean's property window.

The following image displays the PublisherBean properties window in Borland Jbuilder.



Connecting a Publisher to the Broker

Normally, you create a publisher by first creating a PublisherBean. You can set the appropriate ClientID value, and register an exception listener. By default, each message that is written to the PublisherBean is immediately routed to the message broker. However, you can override this default, and queue multiple messages as a unit of work by setting the publisher's *AutoCommit* attribute to FALSE. (The default is TRUE)

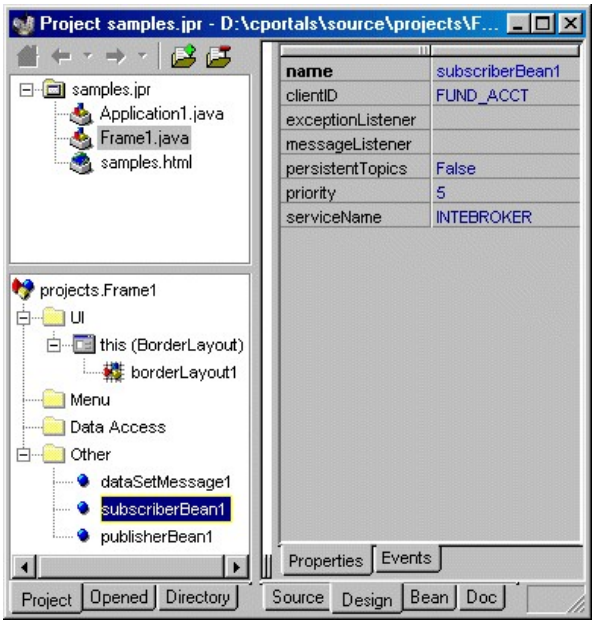
When you call the PublisherBean's `open()` method, the adapter dynamically locates and attaches to the remote message server. Once the publisher is opened, you can iteratively call its `publish(Message)` method. When you have written all the messages for the session, you call the PublisherBean's `close()` method, which flushes any remaining messages and disconnects from the server.

```
PublisherBean publisher = new PublisherBean();
DataSetMessage message = new DataSetMessageBean();
message.setElementCount(10);
...
```

```
publisher.setClientID("MYID");
publisher.setExceptionListener(this);
publisher.setBrokerName("BROKERNAME");
publisher.open();
...
pubisher.publish(mymessage);
...
publisher.close();
```

SubscriberBean Adapter

The SubscriberBean adapter is a threaded component that facilitates performing the functions of a message subscriber. You can subscriber to one or more topics, and indicate whether topical messages should continue to be queued at the server when your application terminates. Like the PublisherBean, the SubscriberBean dynamically locates and attaches to the remote message server. The following image displays the SubscriberBean as it appears in the Borland JBuilder properties window.



Connecting a Subscriber to the Broker

After creating a SubscriberBean object, set the ClientID value to the unique identifier of the subscriber, and register the exception listener object that will be called in the event of a communications error Next, you can register to receive selected topical messages by calling the `addTopic(String)` method.

Whenever the SubscriberBean receives a message from the message server, it passes that message to a registered MessageListener object. Therefore, before connected to the message server, you must register / set an appropriate MessageListener. To begin receiving messages, call the PublisherBean's `open()` method. The `open()` method locates and attaches to the message server, and begins passing messages from the server to the MessageListener.

To terminate receiving messages, call the SubscriberBean's `close()` method.

Note that by default, closing the SubscriberBean also un-subscribes the topics that were added during that subscriber session. If an application chooses to have the session topics continue to be queued by the broker, the client would call the SubscriberBean's setPersistentTopics() method passing a value of TRUE before calling the close() method.

```
SubscriberBean subscriber = new SubscriberBean();
subscriber.setBrokerName("BROKERNAME");
subscriber.setMessageListener(this)
subscriber.setExceptionListener(this);
subscriber.addTopic("TOPIC1");
subscriber.addTopic("TOPIC2");
subscriber.setClientID("MYID");
```

```
subscriber.open();
...
subscriber.close();
```

Sample Code

PublisherBean

In the following example, a PublisherBean is used as the agent for sending messages to the server.

The following code

1. Instantiates a publisher bean and identifies its client ID
2. Opens the publisher bean (makes a connection to the server)
3. Publishes DataSetMessage objects (using the `publish(DataSetMessage)` method) and
4. Closes the publisher bean

```
package osmq.samplecode.basic;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import osmq.clients.Publisher;
import osmq.clients.PublisherBean;
import osmq.util.*;
import osmq.util.ExceptionListener;
import osmq.messages.*;

import java.util.Calendar;
import java.util.Date;

/**
  * Sample code that uses a PublisherBean to attach to a remote
  * message broker and publish a set of topical messages.
  */
public class PubSample implements ExceptionListener
  {
  private static long MAX_PUB = 1000000;
  private long written = 0;
  private ExceptionListener el = null;
  private Publisher publisher;
  private String topicname = "CUSTOMER";
  private boolean isOpen = false;
  DataSetMessage message;

  public PubSample()
    {}
```

```
public void open() throws IOException
  {
  // Create a notification message with 4 elements and a topic of "CLIENT"
  message = MessageFactory.createNotificationMessage(4, "CLIENT");

  // Create a publisher, and set my unique client ID.
  // The 'originator' attribute on messages that I publish
  // will default to this client ID
  publisher = new PublisherBean();
  publisher.setBrokerName(osmq.broker.config.BrokerConsts.DEFAULT_BROKER_SERVICE_NAME);
  publisher.setClientID("MYID");

  // Next I register to be notified regarding any broker-related exceptions.
  // My public function onException(Exception e) will be called in that event.
  publisher.setExceptionListener(this);

  // Open the connection to the message broker.  This performs dynamic
  // discovery and then creates a TCP socket connection to the broker.
  publisher.open();
  }

public void publishAll() throws IOException, MessageException
    {
    while(written++ < MAX_PUB)
        publisher.publish(getNextMessageValues());
    }

public void onException(Exception e)
  {
  System.err.println("Failure event notification: "
                     + e.getMessage());
  this.close();
  System.exit(1);
  }

 // This method is typical of a method that builds the message content.
 private DataSetMessage getNextMessageValues()
    {
    // I indicate the topical event is either an "ADD" "UPDATE"
    // or "DELETE" transaction so that downstream datamarts can be
    // maintained accordingly.
    // This is an optional header field
    message.setTransactionAsChar(MessageAttributes.TRANS_ADD);
```

```
      // I identify the value for the primary (unique) key
      // that is used for table-level database synchronization of
      // downstream datamart maintenance subscribers.
      // This is an optional header field
      message.setKey("PRIMARY_KEY");

      // I clear all former message body values
      message.clearBody();

      // I set each element in the transaction body.
      message.setString(1, "JONE");
      message.setString(2, "FRANCIS");
      message.setDouble(3, 500.00 + getRandom(100000));
      message.setInt(4, 600);

      // Finally, I return the message object
      return message;
      }

  private double getRandom(int multiplier)
       {
       return java.lang.Math.random() * multiplier;
       }

// Closes the publisher bean and disconnects from the broker
private void close()
    {
    try{publisher.close();}
    catch(Exception e){}
    finally{publisher = null;}
    }

public static void main(String[] args)
    {
    PubSample sp = new PubSample();
    try
        {
        sp.open();
        sp.publishAll();
        }
     catch(Exception e){}
     finally{sp.close();}
    }
}
```

BatchPublisher

The BatchPublisher simplifies publishing batch-oriented messages, and delegate the message creation to a descendant of the MessageBuilder class. This can be useful when publishing a series of messages that are derived from extract files.

```
package samplecode.ib;


import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

import osmq.clients.BatchPublisher;
import osmq.util.*;
import osmq.messages.*;

import java.util.Calendar;
import java.util.Date;
import java.sql.*;

public class BatchPubSample implements ExceptionListener, MessageBuilder
  {
  private long written = 0;
  private BatchPublisher cb;
  private DataSetMessage clientmessage;

  public BatchPubSample()
    {
    }

  public String getBuilderID(){return "BATCHPUB";}

  public void open() throws Exception
    {
    // Build a 4-element message for topic "CLIENT"
    clientmessage = MessageFactory.createNotificationMessage(4, "CLIENT");

    // create a batch publisher
    cb = new BatchPublisher("MYID");

    cb.setExceptionListener(this);
    cb.addMessageBuilder(this);
    cb.publishMessages();
    cb.close();
    }

  // this is called by the publisher bean in the event of an exception
  public void onException(Exception e)
    {
    System.err.println("Failure event notification: "
                       + e.getMessage());
    cb.close();
    System.exit(1);
    }



   // This method represents the method that builds the message content.
   // If it returns null, no more messages are to be built, and the application
   // terminates.  Otherwise, the method returns the next message to be published.
   public Message getMessage()
```

```
      {
      if(written++ == 10000)
         return null;
      // I indicate the topical event is an ADD.
      // For subsequent writes that use the same key,
      // the topical event would be an UPDATE
      clientmessage.setTransactionAsChar(MessageAttributes.TRANS_ADD);

      // I identify the value for the primary (unique) key
      // that is used for table-level database synchronization
      clientmessage.setKey("PRIMARY_KEY");



      // I clear all former message body values
      clientmessage.clearBodyElements();

      // I set each element in the transaction body.
      clientmessage.setString(1, "JONE");
      clientmessage.setString(2, "FRANCIS");
      clientmessage.setDouble(3, 500.00);
      clientmessage.setInt(4, 600);
      return clientmessage;
      }


  public static void main(String[] args)
     {
     BatchPubSample sp = new BatchPubSample();

      try
         {
         sp.open();
         }
      catch(Exception e){}
     }
  }
```

SubscriberBean

The SubscriberBean is the preferred class for receiving point-to-point and publish-subscribe messages. It includes dynamic discovery of the message broker, subscription and un-subscription to topics, and a very simplified set of APIs for attaching to and detaching from the broker.

The following code

1. Instantiates a subscriber bean and identifies its client ID
2. Adds a topic subscription request
3. Identifies the method to be called when a message is received by the subscriber bean
4. Opens the subscriber bean (makes a connection to the server)
5. The onMessage() method in the application (the one that receives messages from the bean) is passed messages from the subscriber bean and reads the various message fields based on their position in the message.
6. Finally, based on user input, the subscriber bean is closed, unsubscribing from the session topics and closing the connection to the broker.

```
package osmq.samplecode.basic;

import java.awt.*;
import java.awt.event.*;
import java.io.*;

import osmq.clients.Subscriber;
import osmq.clients.SubscriberBean;
import osmq.util.*;
import osmq.messages.*;
import osmq.util.ExceptionListener;

/**
  * Sample code that uses a SubscriberBean to attach to a remote
  * message broker and retrieve a set of topical messages.
  */
public class SubSample implements ExceptionListener,
                                  MessageListener
  {
  private int ctr = 0;
  private Subscriber bean;
  private boolean isOpen = false;

  public SubSample()
    {
    bean = new SubscriberBean();
    }
```

```
public void open() throws Exception
  {
  bean.setBrokerName(osmq.broker.config.BrokerConsts.DEFAULT_BROKER_SERVICE_NAME);
  // First I identify myself to the message broker by ID and password.  These
  // are mandatory values, to be set before opening the session.
  bean.setClientID("SUBSAMPLE");

  // Subscribe to one or more topics.
  bean.addTopic("CLIENT");

  // A MessageListener is a class with a public onMessage(DataSetMessage) function.
  // Subscribers must identify an instance of MessageListener that will
  // receive topical messages.  Since SampleSubscriber is a MessageListener,
  // this will be registered as the MessageListener
  bean.setMessageListener(this);

  // Next I register to be notified regarding any broker-related exceptions.
  // My public function onException(Exception e) will be called in that event.
  bean.setExceptionListener(this);

  // Connect to the broker, which locates the message server, connects to it,
  // and begins a flow of messages from the broker
  // to my onMessage(DataSetMessage) function.
  bean.open();
  }

/**
  * Public function called by the subcription handler when a DataSetMessage
  * arrives, based either on my subcription to topic(s) (PUB-SUB), or simply
  * addressed to my client ID (POINT-TO-POINT)
  */
public void onMessage(Message ms)
  {
   if(ms.getFormat() != MessageFormat.DATASET)
      throw new IllegalArgumentException("Message type is not dataset");

  // cast the message as a dataset message so I can access the various elements
  DataSetMessage m = (DataSetMessage) ms;

  // Display key fields from every 5000th message
  if((++ctr % 5000) != 0 )
      return;
```

```
    // reference the SSN and last name elements as strings
    System.out.println("Last name is " + m.getString(1));

    System.out.println("First name is " + m.getString(2));

    // reference the salary element as a double
    System.out.println("Salary is "  + m.getDouble(3));

    // reference the units element as an integer
    System.out.println("Shares is "  + m.getInt(4));
    }

  /**
    * Called if there is a serious broker exception.
    */
  public void onException(Exception e)
    {
    byte buffer[] = new byte[10];
    System.out.println("Failure event notification: " + e.getMessage());
    try{System.in.read(buffer);}
    catch(IOException z){}
    close();
    System.exit(1);
    }

  private void close()
        {
        try{bean.close();}
        catch(Exception e){}
        }

  public static void main(String[] args)
    {
    SubSample sample = null;
     try {
          sample = new SubSample();
          sample.open();
          // wait for user to press the enter key
          System.in.read(new byte[100]);
          }
    catch (Exception e)
       {
       e.printStackTrace();
       }
```

```
    finally{
           if(sample != null)
              sample.close();
           System.exit(0);
           }
    }
  }
```