JBoss Messaging User's Guide

A high performance JMS server for JBoss

Table of Contents

1

Introducing JBoss Messaging

JBoss Messaging is a high performance JMS provider in the JBoss Enterprise Middleware Stack (JEMS). It is a complete rewrite of JBossMQ, which is the current default JMS provider in JBoss AS 4.x. JBoss Messaging will be the default JMS provider in JBoss AS 5.x and later, and it is the backbone of the JBoss ESB infrastructure.

Compared with JBossMQ, JBoss Messaging offers vastly improved performance in both single node and clustered environments. Please see this wiki page [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossMessagingPerformanceResultsPre1_0] for performance benchamrks and Chapter 7 on how to generate your own performance benchmarks. It also features a much better internal architecture that would allow us to add more features in the future.

While JBoss Messaging only becomes the default JMS provider from JBoss AS 5.x, production users on JBoss AS 4.x can still take advantage of the performance improvements by easily replacing the JBossMQ module with JBoss Messaging.

In this guide, we discuss how to install and use JBoss Messaging in JBoss 4.x production servers. We cover JBoss Messaging-specific configuration options, as well as how to run the build-in sanity / performance tests.

This guide is work in progress. Please send your suggestions or comments to the JBoss Messaging user forum [http://www.jboss.org/index.html?module=bb&op=viewforum&f=238].

Team:

Ovidiu Feodorov, Project Lead

Tim Fox, Core Messaging Developer

Luc Texier, Lead Support EMEA

Other contributions by: Adrian Brock, Bela Ban, Alex Fu, Aaron Walker

2

Introduction

JBoss Messaging provides an open source and standards-based messaging platform that brings enterprise-class messaging to the mass market.

JBoss Messaging implements a high performance, robust messaging core that is designed to support the largest and most heavily utilized SOAs, enterprise service buses (ESBs) and other integration needs ranging from the simplest to the highest demand networks. JBoss Messaging includes a JMS front-end to deliver messaging in a standards-based format. Additionally, the JBoss Messaging core is also designed to be able to support other messaging protocols in the future.

JBoss Messaging will soon become an integral component of the JBoss Enterprise Middleware Suite (JEMS). Currently it is available for embedded use within the JBoss Application Server (JBossAS), and as a JBoss Microkernel-based stand-alone server. Work to integrate JBoss Messaging with the new JBoss Microcontainer is under way.

The large and vibrant JEMS developer community fosters its continued innovation and enterprise quality. JBoss Messaging enables more agile applications in a wide range of scenarios from simple messaging needs to an enterprise-wide messaging foundation.

JBoss Messaging adds flexibility to any SOA initiative.

2.1. JBoss Messaging 1.0.1 Features

JBoss Messaging provides:

- A fully compatible and Sun certified JMS 1.1 implementation, that currently works with a standard JBossAS 4.x installation and also as a JBoss Microkernel-based standalone deployment.
- A strong focus on performance, reliability and scalability with high throughput and low latency. JBoss Messaging already exceeds JBoss MQ in a number of measured performance metrics. Full results will follow.
- A foundation for JBoss ESB for SOA initiatives; JBoss ESB, due in late 2006, will use JBoss Messaging as its foundation.

JBoss Messaging consists of two major parts:

- JBoss Messaging Core – a transactional, reliable messaging transport system.
    - Supports generalized messages (not just JMS)
    - Enables other messaging protocol façades to be added

    - Distributed, transactional and reliable
- JMS Façade – the JMS "personality" of JBoss Messaging.

Other JBoss Messaging features include:

- Publish-subscribe and point-to-point messaging models
- Topics that feed multiple message queues
- Persistent and non-persistent messages
- Guaranteed message delivery that ensures that messages arrive once and only once
- Transactional and reliable - supporting ACID semantics
- Customizable security framework based on JAAS

2.2. Compatibility with JBossMQ

JBossMQ is the JMS implementation currently shipped within JBossAS. Since JBoss Messaging is JMS 1.1 and JMS 1.0.2b compatible, the JMS code written against JBossMQ will run with JBoss Messaging without any changes.

Important

Even if JBoss Messaging deployment descriptors are very similar to JBoss MQ deployment descriptors, they are *not* identical, so they will require some simple adjustments to get them to work with JBoss Messaging.

3

Download Software

The official releases of JBoss Messaging are available as a free download from the JBoss Messaging project landing page [http://www.jboss.com/products/messaging].

3.1. The JBoss Messaging Release Bundle

The JBoss Messaging release bundle (`jboss-messaging-1.0.x.zip`) will expand in a `jboss-messaging-1.0.x` directory that contains:

- `jboss-messaging-scoped.sar` - the scoped JBoss service archive that contains JBoss Messaging and its dependencies.

  Warning

  Do not simply attempt to copy the archive under a JBoss instance `deploy` directory, since additional steps (such as un-installing JBossMQ and various other configurations tasks) are required for a successful installation. See Chapter 5 for more details.

- `jboss-messaging-client.jar` - the client-side library that need to be in the classpath of the client that opens a remote connection to the Messaging server.

- `util` - a collection of `ant` configuration files used to automate installation and release management procedures. See the Installation chapter for more details.

- `examples` - a collection of examples that should run out of the box and help you validate the installation. Detailed instructions are provided with each example, which range from very simple JMS queue and topic examples to relatively sophisticated use cases in which EJBs and JCA JMS ConnectionFactories are involved. The `examples/config` sub-directory contains various configuration file examples.

- `docs` - this user's guide.

- `src/jboss-messaging-1.0.x-src.zip` - the zipped source directory. The file can be directly installed into and used with a debugger.

  Note

  JBoss Messaging cannot be built using exclusively this source snapshot, which is provided for reference only.

- `src/jboss-messaging-tests-1.0.x-src.zip` - the functional testsuite source archive.

- `test-results` - the output of the functional testsuite, stress and smoke test runs for this release. All these files have been generated during the release procedure.

- `api` - the Messaging API javadoc.
- `README.html` - The release intro document that contains pointers to various other resources, including this Guide.

3.2. CVS Access

If you want to experiment with the latest developments you may checkout the latest code from the 5.0 head branch. Be aware that the information provided in this manual might then not be accurate. For the latest instructions, check out the Messaging Development wiki page [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossMessagingDevelopment].

4

Running the Examples

Since JBoss Messaging is a full JMS provider, it supports all JMS APIs. So, all JMS applications should work without modification. Integrated inside a JBoss AS, we should also access the JMS system from EJBs and write message-driven beans against JMS destinations.

In the following sections, we will look at examples of the various JMS messaging models and message-driven beans. They make use of pre-configured JMS destinations and connection factories that come default with the server. So, no extra configuration is needed to run those examples. Just set JBOSS_HOME and run ANT in each example directory, as we described in Section 5.3. The example source directories are located in the distribution under `docs/examples`.

4.1. Sending messages to a queue

Open an new command line. Set the JBOSS_HOME environment variable to point at a JBossAS 4.x installation. Navigate to the folder where you exploded the main archive and drill down to `/examples/queue`. You need to use Apache Ant to execute the build.xml file. Make sure the JBoss server reference by the JBOSS_HOME is started.

```
public class QueueExample extends ExampleSupport
{

   public void example() throws Exception
   {
      String destinationName = getDestinationJNDIName();

       InitialContext ic = null;
       ConnectionFactory cf = null;
       Connection connection =  null;
       Connection connection2 =  null;

       try {

           ic = new InitialContext();

           cf = (ConnectionFactory)ic.lookup("/ConnectionFactory");
           Queue queue = (Queue)ic.lookup(destinationName);
           log("Queue " + destinationName + " exists");

           connection = cf.createConnection();
           Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
           MessageProducer sender = session.createProducer(queue);

           TextMessage message = session.createTextMessage("Hello!");
           sender.send(message);
           log("The message was successfully sent to the " + queue.getQueueName() + " queue");

           connection2 = cf.createConnection();
           Session session2 = connection2.createSession(false, Session.AUTO_ACKNOWLEDGE);
           MessageConsumer consumer =  session2.createConsumer(queue);
```

```
            connection2.start();

            message = (TextMessage)consumer.receive(2000);
            log("Received message: " + message.getText());
            assertEquals("Hello!", message.getText());

            displayProviderInfo(connection2.getMetaData());

        }catch(NamingException ne){
            ne.printStackTrace();

        }catch(JMSException jmse){
            jmse.printStackTrace();

        }catch(Exception e){
            e.printStackTrace();

        }finally{

            if(ic != null) {
                try {
                    ic.close();
                }catch(Exception ignore){ }
            }

            closeConnection(connection);

            closeConnection(connection2);
        }

    }

    private void closeConnection(Connection con){

        try {
            if (con != null) {
               con.close();
            }

        }catch(JMSException jmse) {
            log("Could not close connection " + con +" exception was " +jmse);
        }
    }


    protected boolean isQueueExample()
    {
       return true;
    }

    public static void main(String[] args)
    {
       new QueueExample().run();
    }

}
```

4.2. Sending messages to a topic

In this example, a standalone Java client publishes a text-based JMS message to a topic and a single subscriber

pulls the message off the queue.

Open an new command line. Set the JBOSS_HOME environment variable to point at a JBossAS 4.x installation. Navigate to the folder where you exploded the main archive and drill down to `/examples/queue`. You need to use Apache Ant to execute the build.xml file Make sure the JBoss server reference by the JBOSS_HOME is started.

```
public class TopicExample extends ExampleSupport
{
   public void example() throws Exception
   {
      String destinationName = getDestinationJNDIName();

       InitialContext ic = null;
       Connection connection = null;

       try {

           ic = new InitialContext();

           ConnectionFactory cf = (ConnectionFactory)ic.lookup("/ConnectionFactory");
           Topic topic = (Topic)ic.lookup(destinationName);
           log("Topic " + destinationName + " exists");

           connection = cf.createConnection();
           Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
           MessageProducer publisher = session.createProducer(topic);
           MessageConsumer subscriber = session.createConsumer(topic);

           ExampleListener messageListener = new ExampleListener();
           subscriber.setMessageListener(messageListener);
           connection.start();

           TextMessage message = session.createTextMessage("Hello!");
           publisher.send(message);
           log("The message was successfully published on the topic");

           messageListener.waitForMessage();

           message = (TextMessage)messageListener.getMessage();
           log("Received message: " + message.getText());
           assertEquals("Hello!", message.getText());

           displayProviderInfo(connection.getMetaData());

       }finally{

           if(ic != null) {
               try {
                   ic.close();
               }catch(Exception e){
                   throw e;
               }
           }

           //ALWAYS close your connection in a finally block to avoid leaks
           //Closing connection also takes care of closing its related objects e.g. sessions
           closeConnection(connection);
       }
   }

   private void closeConnection(Connection con) throws JMSException {

       try {
           if (con != null) {
```

```
            con.close();
          }

       }catch(JMSException jmse) {
           log("Could not close connection " + con +" exception was " +jmse);
           throw jmse;
       }
   }


   protected boolean isQueueExample()
   {
      return false;
   }

   public static void main(String[] args)
   {
      new TopicExample().run();
   }

}
```

4.3. Using JMS from an EJB

This example deploys a simple Stateless Session Bean that is used as a proxy to send and receive JMS messages in a managed environment.

```
public class StatelessSessionExampleBean implements SessionBean
{

   private SessionContext ctx;

    private ConnectionFactory cf = null;

   public void drain(String queueName) throws Exception
   {
      InitialContext ic = new InitialContext();
      Queue queue = (Queue)ic.lookup(queueName);
      ic.close();

      Session session = null;
      Connection conn = null;

      try
      {
         conn = getConnection();
         session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
         MessageConsumer consumer = session.createConsumer(queue);
         Message m = null;
         do
         {
            m = consumer.receiveNoWait();
         }
         while(m != null);
      }
      finally
      {
         closeConnection(conn);
      }
   }
```

```
public void send(String txt, String queueName) throws Exception
{
   InitialContext ic = new InitialContext();
   Queue queue = (Queue)ic.lookup(queueName);
   ic.close();

   Session session = null;
   Connection conn = null;

    try
   {
      conn = getConnection();
      session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

      MessageProducer producer = session.createProducer(queue);

      TextMessage tm = session.createTextMessage(txt);

      producer.send(tm);
      System.out.println("message " + txt + " sent to " + queueName);

   }
   finally
   {
      closeConnection(conn);
   }
}

public List browse(String queueName) throws Exception
{
   InitialContext ic = new InitialContext();
   Queue queue = (Queue)ic.lookup(queueName);
   ic.close();

   Session session = null;
   Connection conn = null;

   try
   {
      conn = getConnection();
      session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
      QueueBrowser browser = session.createBrowser(queue);

      ArrayList list = new ArrayList();
      for(Enumeration e = browser.getEnumeration(); e.hasMoreElements(); )
      {
         list.add(e.nextElement());
      }

      return list;
   }
   finally
   {
      closeConnection(conn);
   }
}

public String receive(String queueName) throws Exception
{
   InitialContext ic = new InitialContext();
   Queue queue = (Queue)ic.lookup(queueName);
   ic.close();

   Session session = null;
   Connection conn = null;
```

```
    try
    {
       conn = getConnection();
       session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

       MessageConsumer consumer = session.createConsumer(queue);

       System.out.println("blocking to receive message from queue " + queueName + " ...");
       TextMessage tm = (TextMessage)consumer.receive(5000);

       if (tm == null)
       {
          throw new Exception("No message!");
       }

       System.out.println("Message " + tm.getText() + " received");

       return tm.getText();

    }
    finally
    {
       closeConnection(conn);
    }
 }

  public Connection getConnection() throws Exception {

      Connection connection = null;

      try {
          connection = cf.createConnection();
          connection.start();

      }catch(Exception e ){
         if(connection != null)
             closeConnection(connection);
         System.out.println("Failed to get connection...exception is " +e);
         throw e;
      }

      return connection;
  }

  public void closeConnection(Connection con) throws Exception {

     try {
         if (con != null) {
             con.close();
         }

     }catch(JMSException jmse) {
         System.out.println("Could not close connection " + con +" exception was " +jmse);
         throw jmse;

     }
  }

 .......
```

4.4. Using EJB2.1 Message Driven Beans

This example deploys a simple Message Driven Bean that processes messages sent to a test queue. Once it receives a message and "processes" it, the MDB sends an acknowledgment message to a temporary destination created by the sender for this purpose. The example is considered successful if the sender receives the acknowledgment message.

The MDB ejb-jar.xml descriptor

```
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>MDBExample</ejb-name>
      <ejb-class>org.jboss.example.jms.mdb.MDBExample</ejb-class>
      <transaction-type>Container</transaction-type>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

The MDB jboss.xml descriptor

```
<enterprise-beans>
  <message-driven>
    <ejb-name>MDBExample</ejb-name>
    <destination-jndi-name>queue/@QUEUE_NAME@</destination-jndi-name>
  </message-driven>
</enterprise-beans>
```

```
public class MDBExample implements MessageDrivenBean, MessageListener
{

   private MessageDrivenContext ctx;

   private ConnectionFactory cf = null;

   public void onMessage(Message m)
   {
      Session session = null;
      Connection conn = null;

      try
      {
         TextMessage tm = (TextMessage)m;

         String text = tm.getText();
         System.out.println("message " + text + " received");
         String result = process(text);
         System.out.println("message processed, result: " + result);

         conn = getConnection();
         session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

         Destination replyTo = m.getJMSReplyTo();
         MessageProducer producer = session.createProducer(replyTo);
         TextMessage reply = session.createTextMessage(result);

         producer.send(reply);
         producer.close();
```

```
      }
      catch(Exception e)
      {
         ctx.setRollbackOnly();
         e.printStackTrace();
         System.out.println("The Message Driven Bean failed!");
      }
      finally
      {
         if (conn != null)
         {
            try {
                closeConnection(conn);
            }catch(Exception e){
                System.out.println("Could not close the connection!" +e);
            }
         }
      }
   }

   private String process(String text)
   {
      // flip the string

      String result = '';

      for(int i = 0; i < text.length(); i++)
      {
         result = text.charAt(i) + result;
      }
      return result;
   }

   public Connection getConnection() throws Exception {

        Connection connection = null;

        try {
            connection = cf.createConnection();
            connection.start();

        }catch(Exception e ){
           if(connection != null)
               closeConnection(connection);
           System.out.println("Failed to get connection...exception is " +e);
           throw e;
        }

        return connection;
    }

    public void closeConnection(Connection con) throws Exception {

       try {
           if (con = null) {
               con.close();
           }

       }catch(JMSException jmse) {
           System.out.println("Could not close connection " + con +" exception was " +jmse);
           throw jmse;
       }
    }

   ......
```

4.5. Using EJB3 Message Driven Beans

This example deploys a simple EJB3 Message Driven Bean that processes messages sent to a test queue. Once it receives a message and "processes" it, the MDB sends an acknowledgment message to a temporary destination created by the sender for this purpose. The example is considered successful if the sender receives the acknowledgment message.

This example relies on having access to a running JBoss Messaging instance. The JBoss Messaging instance must be installed and started according to the "Installation" chapter of this document. The example will automatically deploy its own queue, unless a queue with the same name is already deployed.

This example also relies on having access to the `jboss-messaging-client.jar` archive that comes with the release bundle. If you run this example from an unzipped installation bundle, the example run script is correctly configured to find the client jar. Otherwise, you must modify example's `build.xml` accordingly.

The example was designed to deploy its server-side artifacts under a JBoss' `messaging` configuration. If you intend to use the script with a JBoss configuration that is named differently, please modify the example's `build.xml` accordingly.

Important

The JBoss instance that runs the Messaging server must also have EJB3 support previously installed. If the EJB3 support is not installed, the example will fail with an error message similar to:

```
C:\work\src\cvs\jboss-head\jms\docs\examples\ejb3mdb\build.xml:60: EJB3 does not seem to be installed in
```

For instructions on how to install EJB3 support, please go to JBoss EJB3 documentation page [http://docs.jboss.org/ejb3] or use the JBoss Installer.

The EJB3 Message Driven Bean source code:

```
       @MessageDriven(activateConfig =
       {
             @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue"),
             @ActivationConfigProperty(propertyName="destination", propertyValue="queue/testQueue")
       })
       public class EJB3MDBExample implements MessageListener
       {
          public void onMessage(Message m)
          {
             businessLogic(m);
          }


          private void businessLogic(Message m)
          {
             Connection conn = null;
             Session session = null;

             try
             {
                TextMessage tm = (TextMessage)m;

                String text = tm.getText();
```

```
                System.out.println("message " + text + " received");

                // flip the string
                String result = "";
                for(int i = 0; i < text.length(); i++)
                {
                   result = text.charAt(i) + result;
                }

                System.out.println("message processed, result: " + result);


                InitialContext ic = new InitialContext();
                ConnectionFactory cf = (ConnectionFactory)ic.lookup("java:/JmsXA");
                ic.close();

                conn = cf.createConnection();
                conn.start();
                session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);

                Destination replyTo = m.getJMSReplyTo();
                MessageProducer producer = session.createProducer(replyTo);
                TextMessage reply = session.createTextMessage(result);

                producer.send(reply);
                producer.close();

             }
             catch(Exception e)
             {
                e.printStackTrace();
                System.out.println("The Message Driven Bean failed!");
             }
             finally
             {
                if (conn != null)
                {
                   try
                   {
                      conn.close();
                   }
                   catch(Exception e)
                   {
                      System.out.println("Could not close the connection!" +e);
                   }
                }
             }
          }
       }
```

The basic test examples in this chapter serve as the sanity check for your JBoss Messaging installation. They also provide basic programming examples. To develop your own JMS services, you probably need to configure JBoss Messaging to setup your own destinations and connection factories etc. In Chapter 6, we will discuss JBoss Messaging configuration files and options.

5

Installation

By default, a JBoss AS 4.0.x instance ships with JBossMQ as default JMS provider. In order to use the JBoss AS instance with JBoss Messaging, you need to perform the installation procedure described below.

> **Warning**
>
> A JBossMQ and a JBoss Messaging instance cannot coexist, at least not unless special precautions are taken. Do not simply attempt to copy the Messaging release artifact `jboss-messaging-scoped.sar` over to the JBoss instance `deploy` directory. Follow one of the alternate installation procedures outlined below instead.

5.1. Installing JBoss Messaging with JBoss AS 4.x

5.1.1. Installation procedure

Set up the `JBOSS_HOME` environment variable to point to the JBoss 4.x installation you want to use JBoss Messaging with. Run the installation script, available in the `util` directory of the release bundle. Note that you need Apache Ant 1.6.x or newer installed and accessible from your current directory.

```
cd util
ant -f release-admin.xml
```

The installation script will create a `$JBOSS_HOME/server/messaging` configuration.

> **Note**
>
> If you want to create a JBoss Messaging configuration with a different name, modify the `messaging.config.name` system property declared at the beginning of the installation script accordingly.

5.2. Starting the Server

To run the server, execute the `run.bat` or `run.sh` script as appropriate for your operating system, in the `$JBOSS_HOME/bin` directory.

```
cd $JBOSS_HOME/bin
./run.sh -c messaging
```

A successful JBoss Messaging deployment generates logging output similar to:

```
....
14:23:56,174 WARN  [JDBCPersistenceManager]

JBoss Messaging Warning: DataSource connection transaction isolation should be READ_COMMITTED, but it is
                         Using an isolation level less strict than READ_COMMITTED may lead to data consis
                         Using an isolation level more strict than READ_COMMITTED may lead to deadlock.

14:23:57,276 INFO  [ServerPeer] JBoss Messaging 1.0.1.GA server [server.0] started
14:23:57,937 INFO  [ConnectionFactory] Connector has leasing enabled, lease period 20000 milliseconds
14:23:57,937 INFO  [ConnectionFactory] [/ConnectionFactory, /XAConnectionFactory, java:/ConnectionFactory
14:23:57,987 INFO  [Queue] Queue[/queue/DLQ] started, fullSize=75000, pageSize=2000, downCacheSize=2000
14:23:57,997 INFO  [Topic] Topic[/topic/testTopic] started, fullSize=75000, pageSize=2000, downCacheSize=
14:23:58,017 INFO  [Topic] Topic[/topic/securedTopic] started, fullSize=75000, pageSize=2000, downCacheSi
14:23:58,017 INFO  [Topic] Topic[/topic/testDurableTopic] started, fullSize=75000, pageSize=2000, downCac
14:23:58,027 INFO  [Queue] Queue[/queue/testQueue] started, fullSize=75000, pageSize=2000, downCacheSize=
14:23:58,027 INFO  [Queue] Queue[/queue/A] started, fullSize=75000, pageSize=2000, downCacheSize=2000
14:23:58,037 INFO  [Queue] Queue[/queue/B] started, fullSize=75000, pageSize=2000, downCacheSize=2000
14:23:58,057 INFO  [Queue] Queue[/queue/C] started, fullSize=75000, pageSize=2000, downCacheSize=2000
14:23:58,067 INFO  [Queue] Queue[/queue/D] started, fullSize=75000, pageSize=2000, downCacheSize=2000
14:23:58,077 INFO  [Queue] Queue[/queue/ex] started, fullSize=75000, pageSize=2000, downCacheSize=2000
14:23:58,077 INFO  [Topic] Topic[/topic/openTopic] started, fullSize=75000, pageSize=2000, downCacheSize=
14:23:58,117 INFO  [ConnectionFactoryBindingService] Bound ConnectionManager 'jboss.jca:service=Connectio
....
```

Note

The warning message `DataSource connection transaction isolation should be READ_COMMITTED, but it is currently NONE` is there to remind you that by default JBossAS ships with Hypersonic, an in-memory Java-based database engine, which is apropriate for demo purposes, but not for heavy load production environments. The Critique of Hypersonic [http://wiki.jboss.org/wiki/Wiki.jsp?page=ConfigJBossMQDB] wiki page outlines some of the well-known issues occuring when using this database.

Warning

Before using Messaging in production, you *must* configure the Messaging instance to use an enterprise-class database backend such as MySQL or Oracle, otherwise you risk losing your data. See Section 6.2.1 for details about replacing Hypersonic.

5.3. Installation Validation

The release bundle contains a series of examples that should run "out of the box" and could be used to validate a new installation. Such an example sends a persistent JMS message to a queue called `queue/testQueue`.

To run the example and validate the installation, open an new command line window and set the `JBOSS_HOME` environment variable to point to the JBoss AS 4.x installation you've just installed Messaging on. Navigate to the folder where you extracted the release bundle and drill down to `/examples/queue`. Apache Ant must pe present in your path in order to be able to run the example.

```
setenv JBOSS_HOME=<your_JBoss_installation>
cd .../examples/queue
```

```
    $ant
```

A successfull execution log output looks similar to:

```
Buildfile: build.xml
identify:
  [echo] Running the queue example
  [echo] The queue: testQueue
sanity-check:
init:
compile:
run:
  [java] Queue /queue/testQueue exists
  [java] The message was successfully sent to the testQueue queue
  [java] Received message: Hello!
  [java] The example connected to JBoss Messaging version 1.0.0 (1.0)
  [java] ######################
  [java] ###    SUCCESS!   ###
  [java] ######################
```

It is recommended to run `all` validation examples available in the `example` directory (`queue`, `topic`, `mdb`, `stateless`, etc.). In Chapter 4, we will have a look at each of those examples.

6

Configuration

The JMS API specifies how a messaging client interacts with a messaging server. The exact definition and implementation of messaging services, such as message destinations and connection factories, are specific to JMS providers. JBoss Messaging has its own configuration files to configure services. If you are migrating services from JBossMQ (or other JMS provider) to JBoss Messaging, you will need to understand those configuration files.

In this chapter, we discuss how to configure various services inside JBoss Messaging, which work together to provide JMS API level services to client applications.

Starting with the JBoss Messaging 1.0.1 release, the service configuration is spread among several configuration files. (The 1.0.0 release used to have all configuration information lumped together in the SAR's deployment descriptor `jboss-messaging.sar/META-INF/jboss-service.xml`). Depending on the functionality provided by the services it configures, the configuration data is distributed between `messaging-service.xml`, `remoting-service.xml`, `xxx-persistence-service.xml`, `connection-factories-service.xml` and `destinations-service.xml`.

The AOP client-side and server-side interceptor stacks are configured in `aop-messaging-client.xml` and `aop-messaging-server.xml`.

6.1. Configuring the Server

The Server Peer is the "heart" of the JBoss Messaging JMS facade. The server's configuration, together with the configuration of several core plug-ins (ThreadPool and the MessageStore), resides in `messaging-service.xml` configuration file.

An example of a Server Peer configuration is presented below

```
<mbean code="org.jboss.jms.server.ServerPeer"
    name="jboss.messaging:service=ServerPeer"
    xmbean-dd="xmdesc/ServerPeer-xmbean.xml">

    <constructor>
       <!-- ServerPeerID -->
       <arg type="java.lang.String" value="server.0"/>
       <!-- DefaultQueueJNDIContext -->
       <arg type="java.lang.String" value="/queue"/>
       <!-- DefaultTopicJNDIContext -->
       <arg type="java.lang.String" value="/topic"/>
    </constructor>

    <depends optional-attribute-name="ThreadPool">jboss.messaging:service=ThreadPool</depends>
    <depends optional-attribute-name="PersistenceManager">jboss.messaging:service=PersistenceManager</d
    <depends optional-attribute-name="MessageStore">jboss.messaging:service=MessageStore</depends>
    <depends optional-attribute-name="ChannelMapper">jboss.messaging:service=ChannelMapper</depends>
```

```
    <!-- Set to -1 to completely disable client leasing -->
    <attribute name="SecurityDomain">java:/jaas/messaging</attribute>
    <attribute name="DefaultSecurityConfig">
      <security>
          <role name="guest" read="true" write="true" create="true"/>
      </security>
    </attribute>
  </mbean>
```

6.1.1. SecurityDomain

This identifies the JBoss security domain that will be used when JBoss Messaging authenticates and authorises access to JMS destinations for reading, writing or creating.

It should correspond to a entry in `login-config.xml` where it is configured in exactly the same way as any other security domain in JBoss.

6.1.2. DefaultSecurityConfig

Default security configuration is used when the security configuration for a specific queue or topic has not been overridden in the destination's deployment descriptor. It has exactly the same syntax and semantics as in JBossMQ.

The `DefaultSecurityConfig` attribute element should contain one `<security>` element. The `<security>` element can contain multiple `<role>` elements. Each `<role>` element defines the default access for that particular role.

If the `read` attribute is `true` then that role will be able to read (create consumers, receive messaages or browse) destinations by default.

If the `write` attribute is `true` then that role will be able to write (create producers or send messages) to destinations by default.

If the `create` attribute is `true` then that role will be able to create durable subscriptions on topics by default.

6.2. Configuring Persistence

JBoss Messaging interacts with a persistent store via two services: the Persistence Manager and the Channel Mapper. The Persistence Manager is used to handle the message-related functions. The Channel Mapper manages destination-related persistent data.

JBoss Messaging ships with a JDBC Persistence Manager used for handling persistence of message data in a relational database accessed via JDBC. The Persistence Manager implementation is pluggable (the Persistence Manager is a Messaging server plug-in), this making possible to provide other implementations for persisting message data in non relational stores, file stores etc.

The configuration of "persistent" services is grouped in a `xxx-persistence-service.xml` file, where the actual file prefix is usually inferred from its corresponding database JDBC connection string. By default, Messaging ships with a `hsqldb-persistence-service.xml`, which configures the Messaging server to use the in-VM Hypersonic

database instance that comes by default with any JBossAS instance.

Warning

The default Persistence Manager configuration is works out of the box with Hypersonic, however it must be stressed that Hypersonic should not be used in a production environment mainly due to its limited support for transaction isolation and its propensity to behave erratically under high load.

The Critique of Hypersonic [http://wiki.jboss.org/wiki/Wiki.jsp?page=ConfigJBossMQDB] wiki page outlines some of the well-known issues occuring when using this database.

JBoss Messaging also ships with pre-made Persistence Manager configurations for MySQL, Oracle, PostgreSQL and Sybase. The example `mysql-persistence-service.xml`, `oracle-persistence-service.xml`, `postgres-persistence-service.xml` and `sybase-persistence-service.xml` configuration files are available in the `examples/config` directory of the release bundle.

Configurations for MSSQL and other popular databases should be available soon. Users are encouraged to contribute their own configuration files. The JDBC Persistence Manager has been designed to use standard SQL for the DML so writing a JDBC Persistence Manager configuration for another database is usually only a fairly simple matter of changing DDL in the configuration which is likely to be different for different databases.

The default Hypersonic persistence configuration file is listed below:

```
<server>
   <mbean code="org.jboss.messaging.core.plugin.JDBCPersistenceManager"
      name="jboss.messaging:service=PersistenceManager"
      xmbean-dd="xmdesc/JDBCPersistenceManager-xmbean.xml">
      <depends>jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
      <depends optional-attribute-name="TransactionManager">jboss:service=TransactionManager</depends>
      <depends optional-attribute-name="ChannelMapper">jboss.messaging:service=ChannelMapper</depends>
      <attribute name="DataSource">java:/DefaultDS</attribute>
      <attribute name="CreateTablesOnStartup">true</attribute>
      <attribute name="UsingBatchUpdates">true</attribute>
   </mbean>

   <mbean code="org.jboss.jms.server.plugin.JDBCChannelMapper"
      name="jboss.messaging:service=ChannelMapper"
      xmbean-dd="xmdesc/JDBCChannelMapper-xmbean.xml">
      <depends>jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
      <depends optional-attribute-name="TransactionManager">jboss:service=TransactionManager</depends>
      <attribute name="DataSource">java:/DefaultDS</attribute>
   </mbean>
</server>
```

An example of a Persistence Manager configuration for a MySQL database follows:

```
<server>
   <mbean code="org.jboss.messaging.core.plugin.JDBCPersistenceManager"
      name="jboss.messaging:service=PersistenceManager"
      xmbean-dd="xmdesc/JDBCPersistenceManager-xmbean.xml">
      <depends>jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
      <depends optional-attribute-name="TransactionManager">jboss:service=TransactionManager</depends>
      <depends optional-attribute-name="ChannelMapper">jboss.messaging:service=ChannelMapper</depends>
      <attribute name="DataSource">java:/DefaultDS</attribute>
      <attribute name="CreateTablesOnStartup">true</attribute>
      <attribute name="UsingBatchUpdates">true</attribute>
```

```
      <attribute name="SqlProperties"><![CDATA[
CREATE_MESSAGE_REF=CREATE TABLE JMS_MESSAGE_REFERENCE (CHANNELID BIGINT, MESSAGEID BIGINT, TRANSACTIONID
CREATE_IDX_MESSAGE_REF_TX=CREATE INDEX JMS_MESSAGE_REF_TX ON JMS_MESSAGE_REFERENCE (TRANSACTIONID)
CREATE_IDX_MESSAGE_REF_ORD=CREATE INDEX JMS_MESSAGE_REF_ORD ON JMS_MESSAGE_REFERENCE (ORD)
CREATE_IDX_MESSAGE_REF_MESSAGEID=CREATE INDEX JMS_MESSAGE_REF_MESSAGEID ON JMS_MESSAGE_REFERENCE (MESSAGE
CREATE_IDX_MESSAGE_REF_LOADED=CREATE INDEX JMS_MESSAGE_REF_LOADED ON JMS_MESSAGE_REFERENCE (LOADED)
CREATE_IDX_MESSAGE_REF_RELIABLE=CREATE INDEX JMS_MESSAGE_REF_RELIABLE ON JMS_MESSAGE_REFERENCE (RELIABLE)
INSERT_MESSAGE_REF=INSERT INTO JMS_MESSAGE_REFERENCE (CHANNELID, MESSAGEID, TRANSACTIONID, STATE, ORD, DE
....      ]]>
      </attribute>
      <attribute name="MaxParams">500</attribute>
   </mbean>

   <mbean code="org.jboss.jms.server.plugin.JDBCChannelMapper"
      name="jboss.messaging:service=ChannelMapper"
      xmbean-dd="xmdesc/JDBCChannelMapper-xmbean.xml">
      <depends>jboss.jca:service=DataSourceBinding,name=DefaultDS</depends>
      <depends optional-attribute-name="TransactionManager">jboss:service=TransactionManager</depends>
      <attribute name="DataSource">java:/DefaultDS</attribute>
      <attribute name="SqlProperties"><![CDATA[
CREATE_USER_TABLE=CREATE TABLE JMS_USER (USERID VARCHAR(32) NOT NULL, PASSWD VARCHAR(32) NOT NULL, CLIENT
CREATE_ROLE_TABLE=CREATE TABLE JMS_ROLE (ROLEID VARCHAR(32) NOT NULL, USERID VARCHAR(32) NOT NULL, PRIMAR
SELECT_PRECONF_CLIENTID=SELECT CLIENTID FROM JMS_USER WHERE USERID=?
....
      ]]>
      </attribute>
   </mbean>
</server>
```

6.2.1. Changing the Database

If the database you want to switch to is one of MySQL, Oracle or Postgres, persistence configuration files are already available in the `examples/config` directory of the release bundle.

In order to enable support for one of these databases, just replace the default `hsqldb-persistence-service.xml` configuration file with the database-specific configuration file and restart the server.

Also, be aware that by default, the Messaging services relying on a datastore are referencing `"java:/DefaultDS"` for the datasource. If you are deploying a datasource with a different JNDI name, you need to update all the `Data-Source` attribute in the persistence configuration file.

6.2.2. CreateTablesOnStartup

Set this to `true` if you wish the Persistence Manager to attempt to create the tables (and indexes) when it starts. If the tables (or indexes) already exist a `SQLException` will be thrown by the JDBC driver and ignored by the Persistence Manager, allowing it to continue.

By default the value of `CreateTablesOnStartup` attribute is set to `true`

6.2.3. UsingBatchUpdates

Set this to `true` if the database supports JDBC batch updates. The JDBC Persistence Manager will then group multiple database updates in batches to aid performance.

By default the value of `UsingBatchUpdates` attribute is set to `false`

6.2.4. SQLProperties

This is where the DDL and DML for the particular database is specified. If a particular DDL or DML statement is not overridden, the default Hypersonic configuration will be used for that statement.

6.3. Configuring Destinations

6.3.1. Pre-configured destinations

JBoss Messaging ships with a default set of pre-configured destinations that will be deployed during the server start up. The file that contains configuration for these destinations is `destinations-service.xml`. A section of this file is listed below:

```
<!-- The Dead Letter Queue. This destination is a dependency of an EJB MDB container -->

<mbean code="org.jboss.jms.server.destination.Queue"
       name="jboss.messaging.destination:service=Queue,name=DLQ"
       xmbean-dd="xmdesc/Queue-xmbean.xml">
   <depends optional-attribute-name="ServerPeer">jboss.messaging:service=ServerPeer</depends>
</mbean>

....

<mbean code="org.jboss.jms.server.destination.Topic"
       name="jboss.messaging.destination:service=Topic,name=testTopic"
       xmbean-dd="xmdesc/Topic-xmbean.xml">
   <depends optional-attribute-name="ServerPeer">jboss.messaging:service=ServerPeer</depends>
   <attribute name="SecurityConfig">
      <security>
         <role name="guest" read="true" write="true"/>
         <role name="publisher" read="true" write="true" create="false"/>
         <role name="durpublisher" read="true" write="true" create="true"/>
      </security>
   </attribute>
</mbean>

....

<mbean code="org.jboss.jms.server.destination.Queue"
       name="jboss.messaging.destination:service=Queue,name=testQueue"
       xmbean-dd="xmdesc/Queue-xmbean.xml">
   <depends optional-attribute-name="ServerPeer">jboss.messaging:service=ServerPeer</depends>
   <attribute name="SecurityConfig">
      <security>
         <role name="guest" read="true" write="true"/>
         <role name="publisher" read="true" write="true" create="false"/>
         <role name="noacc" read="false" write="false" create="false"/>
      </security>
   </attribute>
</mbean>

....
```

6.3.2. Destination Configuration Parameters

6.3.2.1. Destination Security Configuration

`SecurityConfig` - allows you to determine which roles are allowed to read, write and create on the destination. It has exactly the same syntax and semantics as the security configuration in JBossMQ destinations.

The `SecurityConfig` element should contain one `<security>` element. The `<security>` element can contain multiple `<role>` elements. Each `<role>` element defines the access for that particular role.

If the `read` attribute is `true` then that role will be able to read (create consumers, receive messaages or browse) this destination.

If the `write` attribute is `true` then that role will be able to write (create producers or send messages) to this destination.

If the `create` attribute is `true` then that role will be able to create durable subscriptions on this destination.

Note that the security configuration for a destination is optional. If a `SecurityConfig` element is not specifed then the default security configuration from the Server Peer will be used.

6.3.2.2. Destination paging parameters

'Pageable Channels' are a sophisticated new feature available in JBoss Messaging.

If your application needs to support very large queues or subscriptions containing potentially millions of messages, then it's not possible to store them all in memory at once.

JBoss Messaging solves this problem but letting you specify the maximum number of messages that can be stored in memory at any one time, on a queue-by-queue, or topic-by-topic basis. JBoss Messaging then pages messages to and from storage transparently in blocks, allowing queues and subscriptions to grow to very large sizes without any performance degradation as channel size increases.

This has been tested with in excess of 10 million 2K messages on very basic hardware and has the potential to scale to much larger number of messages.

The individual parameters are:

`FullSize` - this is the maximum number of messages held by the queue or topic subscriptions in memory at any one time. The actual queue or subscription can hold many more messages than this but these are paged to and from storage as necessary as messages are added or consumed.

`PageSize` - When loading messages from the queue or subscrition this is the maximum number of messages to preload in one operation.

`DownCacheSize` - When paging messages to storage from the queue they first go into a "Down Cache" before being written to storage. This enables the write to occur as a single operation thus aiding performance. This setting determines the max number of messages that the Down Cache will hold before they are flushed to storage.

If no values for `FullSize`, `PageSize`, or `DownCacheSize` are specified they will default to values 75000, 2000, 2000 respectively.

If you want to specify the paging parameters used for temporary queues then you need to specify them on the appropriate connection factory. See connection factory configuration for details.

6.3.3. Deploying a new destination

For a JBoss 4.0.x installation, JBoss Messaging is deployed in its own class loading domain. Because of that you need to deploy a new destinations to use with JBoss Messaging within the same class loading domain.

To deploy a new destination, create a new deployment descriptor named `myqueue-service.xml` (or anything else that ends in `-service.xml`) and copy it to the JBoss instance deployment directory `$JBOSS_HOME/server/messaging/deploy`.

An example of a scoped destination deployment descriptor is listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
 <loader-repository>jboss.messaging:loader=ScopedLoaderRepository
 <loader-repository-config>java2ParentDelegation=false</loader-repository-config>
 </loader-repository>
 <mbean code="org.jboss.jms.server.destination.Queue"
           name="jboss.messaging.destination:service=Queue,name=testQueue"
           xmbean-dd="xmdesc/Queue-xmbean.xml">
           <depends optional-attribute-name="ServerPeer">jboss.messaging:service=ServerPeer</depends>
           <attribute name="SecurityConfig">
              <security>
                 <role name="guest" read="true" write="true"/>
                 <role name="publisher" read="true" write="true" create="false"/>
                 <role name="noacc" read="false" write="false" create="false"/>
               </security>
           </attribute>
           <attribute name="fullSize">75000</attribute>
           <attribute name="pageSize">2000</attribute>
           <attribute name="downCacheSize">2000</attribute>
 </mbean>
</server>
```

6.4. Configuring Connection Factories

With the default configuration JBoss Messaging binds just one connection factory in JNDI at start-up. This connection factory has no client ID and is bound into the following JNDI contexts: `/ConnectionFactory`, `/XAConnectionFactory`, `java:/ConnectionFactory`, `java:/XAConnectionFactory`

You may want to configure additional connection factories, for instance if you want to provide a default client id for a connection factory, or if you want to bind it in different places in JNDI, or if you want different connection factories to use different transports. Deploying a new connection factory is equivalent with adding a new ConnectionFactory MBean configuration to `connection-factories-service.xml`.

It is also possible to create an entirely new service deployment descriptor `xxx-service.xml` altogether and deploy it in `$JBOSS_HOME/server/messaging/deploy`.

An example connection factory configuration is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
   <loader-repository>jboss.messaging:loader=ScopedLoaderRepository
      <loader-repository-config>java2ParentDelegation=false</loader-repository-config>
   </loader-repository>
   <mbean code="org.jboss.jms.server.connectionfactory.ConnectionFactory"
      name="jboss.messaging.destination:service=ConnectionFactory"
      xmbean-dd="xmdesc/ConnectionFactory-xmbean.xml">
      <constructor>
         <arg type="java.lang.String" value="myClientID"/>
      </constructor>
      <depends optional-attribute-name="ServerPeer">jboss.messaging:service=ServerPeer</depends>
      <depends optional-attribute-name="Connector">jboss.messaging:service=Connector,transport=socke
      <attribute name="PrefetchSize">10</attribute>
      <attribute name="DefaultTempQueueFullSize">1000</attribute>
      <attribute name="DefaultTempQueuePageSize">50</attribute>
      <attribute name="DefaultTempQueueDownCacheSize">50</attribute>
      <attribute name="JNDIBindings">
         <bindings>
            <binding>/MyConnectionFactory1</binding>
            <binding>/factories/cf1</binding>>
         </bindings>
      </attribute>
   </mbean>
</server>
```

The above example would create a connection factory with pre-configured client ID `myClientID` and bind the connection factory in two places in the JNDI tree: `/MyConnectionFactory` and `/factories/cf`. The connection factory will use the default remoting connector. To use a different remoting connector with the connection factory change the `Connector` attribute to specify the service name of the connector you wish to use.

prefetchSize is an optional attribute that determines how many messages client side message consumers will buffer locally. Pre-fetching messages prevents the client having to go to the server each time a message is consumed to say it is ready to receive another message. This greatly increases throughput. The default value for prefetchSize is 150. You may want to change this to a smaller value if you are dealing with very large messages, so as not to use too much memory on the client.

DefaultTempQueueFullSize, DefaultTempQueuePageSize, DefaultTempQueueDownCacheSize are optional attributes that determine the default paging parameters to be used for any temporary destinations scoped to connections created using this connection factory. See the section on paging channels for more information on what these values mean. They will default to values of 75000, 2000 and 2000 respectively if ommitted.

6.5. Configuring the remoting connector

JBoss Messaging uses JBoss Remoting for all client to server communication. For full details of what JBoss Remoting is capable of and how it is configured please consult the JBoss Remoting documentation.

The default configuration includes a single remoting connector which is used by the single default connection factory. Each connection factory can be configured to use its own connector.

The default connector is configured to use the remoting socket transport.

This transport opens TCP connections from client to server for client to server communications (e.g. sending messages) and TCP connections from server to client for server to client communications (e.g. receiving messages). The transport can be configured to use SSL where a higher level of security is required.

Future releases JBoss Messaging will support a bidirectional socket transport (similar to UIL2 in JBoss MQ) and an HTTP transport, both of which are useful in network environments where TCP connections from server to client are not possible. This means, for example, that you could deploy one connection factory that uses the HTTP transport for all the connections created from it, and another connection factory that uses the socket transport for all connections created from it.

You can look at remoting configuration under:

<JBoss>/server/<YourMessagingServer>/deploy/jboss-messaging.sar/remoting-service.xml

By default JBoss Messaging binds to ${jboss.bind.address} which can be defined by: ./run.sh -c <yourconfig> -b yourIP.

You can change remoting-service.xml if you want for example use a different communication port, or change any other network behavior.

6.6. Configuring the callback

JBoss Messaging uses a callback mechanism from Remoting that needs a Socket for callback operations. These socket properties are passed to the server by a remote call when the connection is being estabilished. As we said before we will support bidirectional protocols in future releases.

By default JBoss Messaging will execute InetAddress.getLocalHost().getHostAddress() to access your local host IP, but in case you need to setup a different IP, you can define a system property in your java arguments:

Use java -Djboss.messaging.callback.bind.address=YourHost - That will determine the callBack host in your client.

The client port will be selected randomly for any non used port. But if you defined -Djboss.messaging.callback.bind.port=NumericPort in your System Properties that number is going to be used for the call back client port.

7

Generating Performance Benchmark Results

As we discussed in Chapter 1, the key advantage of JBoss Messaging is its superior performance. In fact, the JBoss Messaging comes with a set of standard performance test. You can run them on your server and generate your own performance benchmark results. In this chapter, we will show you how to run a JBoss Messaging server and a JBossMQ server side-by-side on a single machine, and compare their performance. To get the performance tests, you have to obtain the JBoss Messaging source code from CVS as described in Chapter 3.

The test consists in sending bursts of 1000 0 Kilobytes non-persistent messages to both JBoss Messaging and JBossMQ instances while gradually increasing the send rate (200 messages/sec, 400 messages/sec, etc) and measuring the receive rate. At the end, the framework generates the graph representing the receive rate as function of the send rate for two executions (JBoss Messaging and JBossMQ).

7.1. Run JBoss Messaging and JBossMQ Side-by-side

To run performance tests side-by-side on the same machine, we assume that you create two JBoss AS configurations with the JBoss Messaging and JBossMQ modules respectively. We assume that the JBoss Messaging module is installed in the `server/messaging` directory (see Chapter 5), and the default JBossMQ module is installed in `server/jbossmq` directory (just copy the original `default` directory that comes with the server).

Now, if you run the two configurations on the same server, there will be port conflicts. To avoid that, we use the JBoss `ServiceBindingManager` to increase the port numbers in the `jbossmq` configuration by 100 (i.e., the JNDI service will be available at port 1199 instead of 1099). To do that, un-comment the following line in `server/jbossmq/conf/jboss-service.xml`

```
<mbean code="org.jboss.services.binding.ServiceBindingManager"
       name="jboss.system:service=ServiceBindingManager">

  <attribute name="ServerName">ports-01</attribute>
  <attribute name="StoreURL">
    ../docs/examples/binding-manager/sample-bindings.xml
  </attribute>
  <attribute name="StoreFactoryClassName">
    org.jboss.services.binding.XMLServicesStoreFactory
  </attribute>
</mbean>
```

Now, you can start the `messaging` and `jbossmq` configurations side-by-side for testing.

```
run -c messaging
run -c jbossmq
```

7.2. Setup the Tests

The performance framework relies on distributed executors to send messages into the providers being tested. The executors can run standalone in their own VM and act as "remote" JMS connections, or colocated, in which case they are deployed as JBoss services and simulate "colocated" JMS connections.

In order to correctly deploy the colocated executors, the framework relies on the `JBOSS_HOME` environment variable. It assumes directories `$JBOSS_HOME/server/messaging` and `$JBOSS_HOME/server/jbossmq` exist.

```
cd perf
ant sar
ant start-executors
```

Next, we need to deploy test message destinations. They are in the `messaging-destinations-service.xml`, `jbossmq-destinations-service.xml` files. Feel free to add your own destinations (must add equivalent ones in both files) and then deploy them via the following command.

```
ant deploy-destinations
```

7.3. Configure Test Runs

The `perf/perf.xml` file is used to configure tests. In our setting (i.e., `jbossmq` runs in +100 port range from default), the `<providers>` section should look like the following. We can easily run the two JMS server configurations on different machines or in other port ranges. You just need to change the host and port numbers here for tests.

```
<provider name="JBossMessaging">
  <factory>org.jnp.interfaces.NamingContextFactory</factory>
  <url>jnp://localhost:1099</url>
  <pkg>org.jboss.naming:org.jnp.interfaces</pkg>
  <executor name="REMOTE" url="rmi://localhost:7777/standalone"/>
  <executor name="REMOTE2" url="rmi://localhost:7777/standalone2"/>
  <executor name="COLOCATED" url="rmi://localhost:7777/colocated-messaging"/>
  <executor name="COLOCATED2" url="rmi://localhost:7777/colocated-messaging2"/>
</provider>

<provider name="JBossMQ">
  <factory>org.jnp.interfaces.NamingContextFactory</factory>
  <url>jnp://localhost:1199</url>
  <pkg>org.jboss.naming:org.jnp.interfaces</pkg>
  <executor name="REMOTE" url="rmi://localhost:7777/standalone"/>
  <executor name="REMOTE2" url="rmi://localhost:7777/standalone2"/>
  <executor name="COLOCATED" url="rmi://localhost:7777/colocated-jbossmq"/>
  <executor name="COLOCATED2" url="rmi://localhost:7777/colocated-jbossmq2"/>
</provider>
```

The performance configuration section configures how to ramp up the load from 200 messages / sec to 3000 message / sec. We will gather statistics on the number of processed messages versus the number of sent messages.

```
<performance-test name="Throughput 0 KB Message
   Non-Persistent Non-Transactional, 1 sender, 1 receiver">

  <message-size>0</message-size>
  <messages>10000</messages>

  <drain/>

  <parallel>
    <send rate="200" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="400" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="800" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="1000" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="1500" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="2000" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="2500" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <parallel>
    <send rate="3000" executor="COLOCATED"/>
    <receive executor="COLOCATED2"/>
  </parallel>

  <execution provider="JBossMessaging"/>
  <execution provider="JBossMQ"/>

</performance-test>
```

7.4. Run the Tests

To run the tests, simply executes `ant` from the command line. You can access the benchmark result graphs from `output/results/benchmark-results.html`.

After running the test, you can clean up the executors and test destinations using the following commands.

```
ant kill-executors
ant undeploy-destinations
```