RM4GS Programming Guide

version 1.0

September 2004

Abstract

This document describes programming guide of RM4GS (Reliable Messaging for Grid Services).

1. Abstract

This chapter describes how to run RM4GS applications.

1.1. Creating Queues

RM4GS applications obtain ConnectionFactory objects and Destination objects, that represent sending or receiving destinations, through JNDI. These objects are used to send or receive messages. To run the application, these objects must be created and registered with JNDI before the application can use them. ConnectionFactory objects, will have been created during the installation process as described in the RM4GS Install Guide. If RM4GS has been installed according to that document, a ConnectionFactory object will be registered with JNDI as eis/rm4gs by the command asadmin create-connector-resource. Destination objects must be created for each application.

First, create a physical queue for the sender and receiver. Use the following command.

```
> create-queue -qname SimpleQueue -qtype VOLATILE
```

The above command creates the physical queue named “SimpleQueue” on the RM4GS managed database. Second, execute the following command to associate the physical queue with a Destination object that can be used by applications. The command must be executed in a single line, please ignore line breaks in the examples below. The name “eis/SimpleQueue” is specified as the JNDI name in this example.

```
> asadmin create-admin-object --user admin --restype
org.bizgrid.rm4gs.P2PDestination --raname rm4gs --property
Destination=SimpleQueue eis/SimpleQueue
```

1.2. Example Code

The following example demonstrates a message sender application client. Specify the JNDI name of the ConnectionFactory and the JNDI name of the queue respectively in the hatched part of the code. While the following code specifies the JNDI name used at creation time, you can change the JNDI mapping freely by prefixing java:comp/env.

```
    main(String[]args) {
        InitialContext ctx = new InitialContext();
        // Obtain  a Connection.
        // Specify the JNDI name of the connection factory.
        ConnectionFactory cf =
               (ConnectionFactory)ctx.lookup("eis/rm4gs");
        Connection conn = cf.getConnection(true, 0);

        // Obtain a P2Pdestination. Specify the name of the queue.
        P2PDestination dest = (P2PDestination)ctx.lookup("eis/SimpleQueue");

        // Create a sending session.
        Policy[] policies = new Policy[] { Policy.EXACTLY_ONCE, };
        SendingSession session = conn.createSendingSession(dest, policies);

        // Start the RM4GS transaction.
        conn.begin();

        // Create a message.
        TextMessage msg1 = (TextMessage)conn.createMessage(MessageType.TEXT);

        msg1.setMessage(...);

        // Send the message.
        session.send(msg1);


        // Complete the RM4GS transaction.
        conn.commit();

        // Release the message.
        msg1.release();

        // Release the session.
        session.close();

        // Release the connection.
        conn.close();
}
```

The following is an example of an MDB code that receives messages. There is no information such as receiving queue names in the MDB code. Such information is specified in the MDB deployment descriptor.

```
public class MessageBean implements MessageDrivenBean, MessageListener {

    private MessageDrivenContext mdc = null;

    public MessageBean() {
    }

    public void setMessageDrivenContext(MessageDrivenContext mdc) {
        this.mdc = mdc;
    }

    public void ejbCreate() {
    }

    public void ejbRemove() {
    }

    public void onMessage(Message msg) {
        TextMessage textmsg = null;

        try {
            if (msg instanceof TextMessage) {
                textmsg = (TextMessage) msg;
                System.out.println("Got Message = " + textmsg.getMessage());
            } else {
                System.out.println("Unknown type of Message received");
            }
        } catch (RM4GSException e) {
            e.printStackTrace();
        }
    }
}
```

The following demonstrates an MDB deployment descriptor. The hatched part specifies a queue name Parameters such as MaxThreads are also specified here. See the RM4GS Reference Guide for details of the parameters.

```
<ejb-jar...>
    <display-name>MessageBean</display-name>
    <enterprise-beans>
        <message-driven>
            <ejb-name>MessageBean</ejb-name>
            <ejb-class>MessageBean</ejb-class>
            <messaging-type>
                org.bizgrid.rm4gs.MessageListner
            </messaging-type>
            <transaction-type>Bean</transaction-type>
            <activation-config>
                <activation-config-property>
                    <activation-config-property-name>
                        Destination
                    </activation-config-property-name>
                    <activation-config-property-value>
                        SimpleQueue
                    </activation-config-property-value>
                </activation-config-property>
                <activation-config-property>
                    <activation-config-property-name>
                        MaxThreads
                    </activation-config-property-name>
                    <activation-config-property-value>
                        1
                    </activation-config-property-value>
                </activation-config-property>
            </activation-config>
        </message-driven>
    </enterprise-beans>
</ejb-jar>
```

1.3. Communication between two machines

To communicate between two machines, queues must be created on each machine. To send messages from machine A to machine B, specify the address of the forwarding machine when creating a queue on machine A. The following demonstrates how to create a queue on machine A.

```
> create-queue -qname ForwardQueue –forward http://<machineB>:<port>/rm4gs/
-qtype VOLATILE
>      asadmin      create-admin-object      --user      admin      --restype
org.bizgrid.rm4gs.P2PDestination       --raname       rm4gs       --property
Destination=ForwardQueue eis/ForwardQueue
```

The create-queue command is different from the single machine case where the forwarding machine address is specified by the option –forward. The command asadmin create-admin-object is the same as for the single machine case.

Create a queue on the receiving machine B using the same command as for the single machine case. Specify the same queue name used at the sender. Execute the following command on machine B.

```
> create-queue -qname ForwardQueue -qtype VOLATILE
>      asadmin      create-admin-object      --user      admin      --restype
org.bizgrid.rm4gs.P2PDestination       --raname       rm4gs       --property
Destination=ForwardQueue eis/ForwardQueue
```

The code is little different from the single machine case. It has to look up the JNDI name “eis/ForwardQueue” that was specified above when looking up a Destination object through JNDI.

2. Creating Destinations in Another Way

In the previous section, a P2Pdestination object was first registered with JNDI, and then the application code looked up it through JNDI. In some cases, however, the application must specify an explicit address. RM4GS supports this case so that a P2Pdestination may be specified as the address of the forwarding machine.

```
main(String[]args) {
    InitialContext ctx = new InitialContext();
    // Obtain a Connection.
    // Specify the JNDI name of the connection factory.
    ConnectionFactory cf =
           (ConnectionFactory)ctx.lookup("eis/rm4gs");
    Connection conn = cf.getConnection(true, 0);

    // Create a P2PDestination.
    URL url = new URL("http://machine B:port/rm4gs/ForwardQueue");
    P2PDestination dest = conn.createP2PDestination(url);
```

Create a URL object by specifying the forwarding address. The URL is the address of the RM4GS on the forwarding machine. The last “/” separated element should be a queue name. A P2PDestination object may be specified by passing the URL to the method “createP2PDestination” of the Connection object. You can use the Destination object in the same way as the code previously demonstrated.

3. Advanced Usage

This section describes advanced RM4GS functionality.

3.1. Transactions

RM4GS can send and receive multiple messages in a transaction. Consider the case of an application that sends messages to multiple destinations. The desired behavior may be that all destinations receive their message or no

destinations receive any message due to some error. The desired result may be obtained by sending messages in a transaction.

For example;

```
For each destination i {
        Some work for the destination i                  (1)
        Create a message for the destination i.
        Send the message to the destination i.           (2)
}
```

Consider the case that an error has occurred during the work (1) for the second destination and the code cannot continue the work. Since the work (2) for the first destination has already been completed, some handling such as sending a cancellation message to the first destination is necessary. In this case, complicated error handling is not necessary if all works in the iteration are executed as a transaction.

```
Start the transaction.
For each destination i {
        Some work for the destination i                  (1)
        if (error has occurred) Abort the transaction.
        Create a message for the destination i.
        Send the message to the destination i.           (2)
}
Commit the transaction.
```

When an RM4GS application sends messages in a transaction, the messages are not actually sent until the application commits the transaction. On commit, all queued messages are actually processed and sent. On abort, queued messages are discarded. The above code is simplified in this way.

Note that in a transaction, messages are not actually sent until commit. Thus it is impossible to send a message to a destination and receive a reply message from the destination in a same transaction.

Transactions can be applied not only to sending work but also to receiving work.


```
{
        Start the transaction.
        Receive a message.                  (1)
        Process the message.
        if (error has occurred) {
                Abort the transaction;      (2)
                break;
        }
        Receive a message.
        Process the message.                (3)
        Commit the transaction.
}
{
        Receive a message.                  (4)
}
```

When the transaction is aborted at (2) in the above code, the message received at (1) is received at (4) again. It is because RM4GS cancels the work (1).

The following shows a transactional example using RM4GS.

```
main(String[]args) {
    InitialContext ctx = new InitialContext();
    // Obtain a Connection.
    // Specify the JNDI name of the connection factory.
    ConnectionFactory cf =
           (ConnectionFactory)ctx.lookup("eis/rm4gs");
    Connection conn = cf.getConnection(true, 0);     // (1)

    // Obtain a P2PDestination. Specify the queue name.
    P2PDestination dest = (P2PDestination)ctx.lookup("eis/SimpleQueue");

    // Create a sending session.
    Policy[] policies = new Policy[] { Policy.EXACTLY_ONCE, };
    SendingSession session = conn.createSendingSession(dest, policies);

    // Start the RM4GS transaction.
    conn.begin();        // (2)

    // Create a message.
    TextMessage msg1 = (TextMessage)conn.createMessage(MessageType.TEXT);

    msg1.setMessage(...);

    // Send the message.
    session.send(msg1);


    // Complete the RM4GS transaction.
    conn.commit();       // (3)

    // Release the message.
    msg1.release();

    // Release the session.
    session.close();

    // Release the connection.
    conn.close();
}
```

To use transactions, specify true in the first argument of “getConnection” when obtaining a Connection object (at (1) on the above code). This declares that transactional calls will be made on the obtained Connection object. When the first argument is true, the second argument must be 0. (The second argument is used to specify the behavior when transaction is not used.)

To start a transaction, call the method “begin” on a Connection object (the above (2)). To commit the transaction, call the method “commit” on the Connection object (the above (3)). To abort the transaction, call the method “abort”.

3.2. Guaranteed Message Ordering

Guaranteed Message Ordering is a feature that ensures that messages are received in the same order that they were sent. The sender specifies Guaranteed Message Ordering by sending messages with Guaranteed Message

Ordering QoS. The receiver has no role in specifying Guaranteed Message Ordering.. The following example demonstrates a sending application using Guaranteed Message Ordering.

```
    main(String[]args) {
        InitialContext ctx = new InitialContext();
        // Obtain a Connection.
        // Specify the JNDI name of the connection factory.
        ConnectionFactory cf =
               (ConnectionFactory)ctx.lookup("eis/rm4gs");
        Connection conn = cf.getConnection(true, 0);

        // Obtain a P2PDestination. Specify the queue name.
        P2PDestination dest = (P2PDestination)ctx.lookup("eis/SimpleQueue");

        // Create a sending session.
        Policy[] policies = new Policy[] { Policy.IN_ORDER, };
        SendingSession session = conn.createSendingSession(dest, policies);

        // Start the RM4GS transaction.
        conn.begin();

        for (int i=0;i<10;i++) {
            // Create a message.
            extMessage msg =
                (TextMessage)conn.createMessage(MessageType.TEXT);

            msg.setMessage("Message"+i);
            // Send the message.
            session.send(msg);
            // Release the message.
            msg.release();
        }
        // Complete the RM4GS transaction.
        conn.commit();


        // Release the session.
        session.close();

        // Release the connection.
        conn.close();
}
```

The sending application invokes “Connection#createSendingSession” by specifying the policy “Policy.IN_ORDER” to create a “SendingSession”. Guaranteed Message Ordering is specified by invoking the method “send” on the “SendingSession” object. Guaranteed Message Ordering is effective until the “SendingSession” is closed.

3.3. Guaranteed Message Ordering beyond the application’s lifetime

In the previous section, Guaranteed Message Ordering was in use for messages sent before “SendingSession” was closed. “SendingSession” cannot be kept open for a sender that terminates and then restarts. In some cases, Guaranteed Message Ordering is also necessary for messages sent by such sender. You can use “ReusableSendingSession” instead of “SendingSession” for this case. By specifying a group ID explicitly when

creating a" ReusableSendingSession", Guaranteed Message Ordering is in use for messages belonging to the group beyond the application's lifetime.

```
// Create a sending session for Guaranteed Message Ordering beyond the
application's lifetime.

Policy[] policies = new Policy[] {
       Policy.IN_ORDER,
       Policy.createGroupId("myorder")};
ReusableSendingSession session =
       conn.createReusableSendingSession(dest, policies);
```

A group ID must be explicitly specified when creating a "ReusableSendingSession". Guaranteed Message Ordering is in use for all messages containing the group ID. You can create a group ID by using the method "Policy.createGroupId". Any string may be specified for group ID, but you have to be sure that the string is not the same as any other group ID. If the same group ID is accidentally reused, Guaranteed Message Ordering may occur for messages sent over some other unintended "ReusableSendingSession" using the same group ID. Note that RM4GS itself uses group IDs implicitly. The Group IDs that RM4GS automatically assigns are given a specific prefix. Unintentional collision may be avoided by using a different prefix for application generated group IDs. See the RM4GS Reference Guide for the prefix of group IDs that RM4GS implicitly uses. ReusableSendingSession must be explicitly removed when it is unnecessary. The following shows how to remove it.

```
// Remove ReusableSendingSession.

Policy[] policies = new Policy[] {
       Policy.IN_ORDER,
       Policy.createGroupId("myorder")};
ReusableSendingSession session =
       conn.createReusableSendingSession(dest, policies);
session.delete();
```

First, create a ReusableSendingSession by specifying a group ID that you want to remove. You can then remove the group ID by invoking the method "delete" on the "ReusableSendingSession" object.

3.4. Using J2EE Message Driven Beans

The MDB called back by RM4GS implements the RM4GS message listener interface.

The following example code demonstrates this.


```
import javax.ejb.*;
import org.bizgrid.rm4gs.*;

public class MessageBean implements MessageDrivenBean, MessageListener {
   private MessageDrivenContext mdc;

   public MessageBean() {
      super();
   }

   public void setMessageDrivenContext(MessageDrivenContext mdci)
      throws EJBException {
      mdc = mdci;
   }

   public void ejbCreate() {}

   public void ejbRemove() {}

   public void onMessage(Message msg) { // (1)
      TextMessage textmsg = null;

      try {
         if (msg instanceof TextMessage) {
            textmsg = (TextMessage) msg;
            System.out.println("Got Message = " + textmsg.getMessage());
         } else {
            System.out.println("Unknown type of Message received");
         }
      } catch (RM4GSException e) {
         e.printStackTrace();
      }
   }
}
```

The method “onMessage” at (1) in the above code is called back by RM4GS. The argument “msg” holds the received data.

The following is the deployment descriptor for the above code.

```
<ejb-jar...>
    <display-name>MessageBean</display-name>
    <enterprise-beans>
        <message-driven>
            <ejb-name>MessageBean</ejb-name>
            <ejb-class>MessageBean</ejb-class>   // (1)
            <messaging-type>
                org.bizgrid.rm4gs.MessageListner   // (2)
            </messaging-type>
            <transaction-type>Bean</transaction-type>
            <activation-config>
                <activation-config-property>
                    <activation-config-property-name>
                        Destination
                    </activation-config-property-name>
                    <activation-config-property-value>
                        SimpleQueue   // (3)
                    </activation-config-property-value>
                </activation-config-property>
                <activation-config-property>
                    <activation-config-property-name>
                        MaxThreads
                    </activation-config-property-name>
                    <activation-config-property-value>
                        1   // (4)
                    </activation-config-property-value>
                </activation-config-property>
            </activation-config>
        </message-driven>
    </enterprise-beans>
</ejb-jar>
```

In the above deployment descriptor, specify the MDB class at (1), the RM4GS message listener interface at (2), and destination's logical name at (3). Specify the multiplicity when MDB is called back at (4). If you specify 5, for example, five MDB instances are created and five threads invoke the each MDB instance.

