FioranoMQ® 9

C++ RTL(JNI) Guide

Contents

Chapter 1: Overview

The FioranoMQ server implements a C++ runtime library, allowing C++ programs to use the JMS messaging facilities through a thin set of wrapper classes. C++ calls are translated to Java calls using the Java Native Interface (JNI). By using the C++ API for JMS, C++ and Java programs can exchange data seamlessly. Specification on the Sun Microsystems website.

Related Documentation

For complete information on the available Runtime libraries, we strongly recommend that you go through the entire range of FioranoMQ RTL Documentation.

Document Name	Description
C Runtime Library Guide	Contains the description of the FioranoMQC Runtime APIs.
Native C ++ Runtime Library Guide	Contains a description of FioranoMQ native C++ Runtime APIs.
Native C# Runtime Library Guide	Contains a description of the FioranoMQ C# Runtime APIs.
JavaDocs	Contains "javadoc" style documentation on all public Fiorano classes.

Table 1: Related Documentation

Chapter 2: Using FioranoMQ C++ Runtime Library

C++ wrappers are provided for the complete set of standard JMS APIs. Both Pub/ Sub and PTP APIs are covered, allowing C++ programs to publish and subscribe to both Topics and Queues. C++ wrappers are provided only for the core JMS APIs and not for the Administration and Security classes. To create administered objects, you should install and use the AdminTool that is supplied with the deployment edition of the FioranoMQ server.

Detailed documentation on the C++ APIs is available in HTML files in the docs/ directory of the FioranoMQ installation. The C++ APIs are exactly analogous to the Java APIs and are used in precisely the same manner. However, since C++ does not support automatic garbage collection, the C++ programmer must be careful to free up allocated memory by deleting objects when they go out of scope.

Using the Application Programming Interface

Admin Objects Lookup

The lookup () operations on administered objects such as topics, queues, connection factories etc can be achieved using the APIs provided by InitialContext. The following code snippet illustrates the usage of InitialContext to lookup administered objects.

```
Hashtable *table = new Hashtable();
table->put(SECURITY_PRINCIPAL, m_usrName);
table->put(SECURITY_CREDENTIALS, m_usrPasswd);
table->put(PROVIDER_URL, m_providerURL);
table->put(INITIAL_CONTEXT_FACTORY,m_initialContextFactory);

m_ic = new InitialContext (table);

m_tcf =(TopicConnectionFactory*) m_ic->lookup (m_tcfName);

m_topic =(Topic*) m_ic->lookup (m_topicName);

cout << ">>Lookedup admin objects " << m_tcfName << " and "
<< m_topicName << "\n" <<flush;
}
```

Creating the Connection and Setting an Exception Listener

The following code sample illustrates the process of creating a connection and setting an ExceptionListener over the connection. This allows a JMS provider, to informs the connection's ExceptionListener if it detects a serious problem with a connection. It does this by calling the listener's onException method, passing it a JMSException object describing the problem.

```
// Exception listener
class MyEListener : public ExceptionListener
{
public:
MyEListener () : ExceptionListener () {}
void onException (JMSException* je)
{
ex->printStackTrace();
}
};

class MyListener : public MessageListener

{
public:
MyListener () : MessageListener ()
{
}

void onMessage (Message* msg)
{

try

{
    TextMessage* tMsg = (TextMessage *)msg;
    char* str = tMsg->getText ();
    cout << "received message : " << str << endl;

delete str;
}
catch (...)
{

    cerr << "Error reading message" << endl;
}
 }
};

       TopicConnectionFactory* m_tcf =
(TopicConnectionFactory*) m_ic->lookup (m_tcfName);

cout << "Creating topic connection" << endl;
m_tc = m_tcf->createTopicConnection
(m_usrName,m_usrPasswd);
m_tc->start();
```

```
ExceptionListener* exListener = new MyEListener ();
m_tc->setExceptionListener (exListener);

      cout << "Creating topic session." << endl;
m_ts = m_tc->createTopicSession (0, 1);
cout << "Creating topic subscriber on primaryTopic." <<

endl;
m_subscriber =m_ts->createSubscriber (m_topic, NULL,
false);
  m_subscriber->setMessageListener (new MyListener ());
```

Using C++ APIs for Publishing Messages

In this section, we examine a complete C++ program that publishes messages to the topic “primaryTopic”. Any Java or C++ program that subscribes to this topic can pick up messages published by the C++ publisher.

Example: C++ program to publish messages to the topic “primaryTopic"

```
#include "jms.h"
#include "iostream.h"

class Publisher
{
InitialContext* m_ic;

public:

void runPublisher ()

{
// Create the InitialContext Object
// New :: From FMQ 5.0 upwards

//The older version FioranoInitialContext is also
//been supported for backward compatibility reasons.

     Hashtable* table = new Hashtable();

    table->put (SECURITY_PRINCIPAL, "anonymous");
    table->put (SECURITY_CREDENTIALS, "anonymous");
    table->put (PROVIDER_URL, "http://localhost:1856");

    table->put (INITIAL_CONTEXT_FACTORY,
       "fiorano.jms.runtime.naming.FioranoInitialContextFactory");

    m_ic = new InitialContext (table);

    // Look up for the primary topic connection factory
    TopicConnectionFactory* tcf =
    (TopicConnectionFactory*) m_ic->lookup ("prima-
ryTCF");
```

```
    Topic* topic =
    (Topic*) m_ic->lookup ("primarytopic");

    // Create and start the Topic connection
    TopicConnection* tc = tcf->createTopicConnection ();

    tc->start ();

    // Create the topic session
    TopicSession* ts = tc->createTopicSession (0, 1);

    // Create the publisher on the primary topic
    TopicPublisher* tp = ts->createPublisher (topic);

    // Read input from STDIN and publish a message

    // to primaryTopic.
    while (true)
 {

TextMessage* msg = ts->createTextMessage ();
char* input = new char [1024];
cout << "Enter a message To be published: ";

cin.getline (input, 1024, ' ');
msg->setText (input);

// Publish a Non-Persistent message (with default
// priority and infinite TTL (Time To Live).
tp->publish (topic, msg, 1, 0, 0);

delete[] input;
delete msg;
 }

    tc->close ();
}

~Publisher ()
 {
    // Free up all memory held up by the InitialContext.
    if (m_ic != NULL)
delete (m_ic);
 }

};

// Main program creates a publisher on “primaryTopic” and
// starts it up
void main (char** args)
{
```

```
try
{
Publisher pub;
pub.runPublisher ();

}
catch (JMSException* ex)
{

cout << "Error while running Publisher " << endl;
ex->printStackTrace();
delete ex;

}
}
```

The runPublisher member function in the Publisher class incorporates major functionality of the program. The main program in the file Publisher.cpp creates a single Publisher object and starts publishing messages by invoking the runPublisher() method.

As can be seen from the program, the C++ JMS wrappers are used in much the same way as their Java counterparts. For instance, the calls

```
TopicConnectionFactory* tcf = (TopicConnectionFactory*) m_ic-
>lookup ("primryTCF");

Topic* topic = (Topic*) m_ic->lookup ("primarytopic");
```

are used to lookup TopicConnectionFactories and Topics, respectively. The only difference between the above C++ calls and equivalent Java calls is the use of the pointer syntax in C++ in comparison to object dereferencing in Java.

Similarly, the calls

```
// Create a Topic Connection
TopicConnection* tc = tcf->createTopicConnection ();

// Create a Topic Session
TopicSession* ts = tc->createTopicSession (0, 1);

// Create a Publisher on a Topic
TopicPublisher* tp = ts->createPublisher (topic);
```

are used to create a TopicConnection, TopicSession and TopicPublisher, respectively. The equivalent Java calls would be:

```
TopicConnection tc = tcf.createTopicConnection ();
TopicSession ts = tc.createTopicSession (0, 1);
TopicPublisher tp = ts.createPublisher (topic);
```

Note:The only difference between the C++ and Java programs is that the C++ versions use pointers rather than objects.

Here, the loop

```
while (true)
{
TextMessage* msg = ts->createTextMessage ();
char* input = new char [1024];
cout << "Enter a message To be published: ";
cin.getline (input, 1024, '');
msg->setText (input);

// Publish a Non-Persistent message (with default
// priority and infinite TTL (Time To Live).
tp->publish (topic, msg, 1, 0, 0);

delete[] input;
delete msg;
}
```

reads input strings from standard input and publishes them to the topic primaryTopic.

Note: Unlike Java, the C++ program needs to explicitly delete the allocated storage for both the message msg as well as the temporary buffer input.

C++ Program to Listen for Messages on a Topic

The following program implements an asynchronous listener to listen for messages published on the topic primaryTopic. The messages published by the program Publisher.cpp can be received using this program.

```
// Implements an asynchronous listener to listen
// for messages published on the topic "primaryTopic"
//
#include "jms.h"
#include "iostream.h"

class MyListener : public MessageListener

{
public:
MyListener () : MessageListener () {}

void onMessage (Message* msg)

 {
 try
 {

  TextMessage* tMsg = (TextMessage *)msg;
  char* str = tMsg->getText ();

  cout << "received message : " << str << endl;
  delete str;
```

```
 }
     catch (JMSException* ex)
 {

  cerr << "Error reading message" << endl;
  ex->printStackTrace();
       delete ex;
}
}
};


// Exception listener to reconnect to server in case
// of server shutdown
class MyEListener : public ExceptionListener
{

public:
MyEListener () : ExceptionListener () {}

void onException (JMSException* je)
{
     ex->printStackTrace();

}
};

class Subscriber
{
InitialContext* m_ic;

public:

void runSubscriber ()
{
     Hashtable* table = new Hashtable();
     table->put (SECURITY_PRINCIPAL, "anonymous");

        table->put (SECURITY_CREDENTIALS, "anonymous");
     table->put (PROVIDER_URL, "http://localhost:1856");
     table->put (INITIAL_CONTEXT_FACTORY,

"fiorano.jms.runtime.naming.FioranoInitialContextFactory");
     m_ic = new InitialContext (table);

    // create the topic connection factory
   TopicConnectionFactory* tcf =
(TopicConnectionFactory*) m_ic->lookup ("primaryTCF");

   Topic* topic = (Topic*) m_ic->lookup ("primarytopic");
```

```
   // create the topic connection
   TopicConnection* tc = tcf->createTopicConnection ();
   tc->start ();

   // create and set an exception listener for this
   // connection
   ExceptionListener* exListener = new MyEListener ();

    tc->setExceptionListener (exListener);

// create the topic session
       TopicSession* ts = tc->createTopicSession (0, 1);

// create the topic subcriber
       TopicSubscriber* tsub = ts->createSubscriber (topic,
NULL, false);

// set the message listener on the subscriber
 tsub->setMessageListener (new MyListener ());

       // Wait for program to terminate.
       int dummy;
       cin >> dummy;
       tc->stop ();

}

~Subscriber ()

 {
if (m_ic != NULL)
delete (m_ic);

}
};

void main (char** args)

{
try
{

Subscriber sub;
sub.runSubscriber ();
}
    catch (JMSException* ex)
 {

cout << "Error while running Subscriber " << endl;
ex->printStackTrace();
delete ex;

 }
}
```

Multithreaded Clients

To develop a multithreaded application, you can either use _beginthreadex() API provided by Win32 platforms or pthread_create() API. The code snippet given below illustrates the usage of these APIs.

```
#include <JMS.h>
#include <iostream.h>
#ifdef _WIN32
#include <process.h>
#include <windows.h>
#else
#include <pthread.h>
#include <stdlib.h>
#endif
#include <jni.h>
#ifdef _WIN32
typedef unsigned int (*MyRunnable)(void*);


#else
typedef void * (*MyRunnable)(void*);
#endif


class MyListener : public MessageListener
{
public:
MyListener () : MessageListener ()
{
}

void onMessage (Message* msg)


{
try
{

TextMessage* tMsg = (TextMessage *)msg;
char* str = tMsg->getText ();
printf("Received message :: %s\n",str);
delete str;


}
catch (JMSException* jex)
{


jex->printStackTrace();
delete jex;
```

```
}
}
};

#ifdef _WIN32
unsigned int newthread(void* arg)
#else
void newthread(void* arg)
#endif
{
try
{

InitialContext* new_ic = (InitialContext*)arg;
TopicConnectionFactory* tcf =
(TopicConnectionFactory*) new_ic->lookup ("primaryTCF");


Topic* topic = (Topic*) new_ic->lookup ("primarytopic");


cout << "Creating topic connection" << endl;
TopicConnection* tc = tcf->createTopicConnection ();
tc->start ();
cout << "Creating topic session" << endl;
TopicSession* ts = tc->createTopicSession (0, 1);
cout << "Creating topic publisher on primary Topic" << endl;
TopicPublisher* tp = ts->createPublisher (topic);
cout << "Messages are being  published to primary Topic: "<<endl;
cout << "Enter Q to quit: "<<endl;

while (true)
{
TextMessage* msg = ts->createTextMessage ();
char* input = new char [1024];
cout << "Enter a message To be published: ";


cin.getline (input, 1024, '\n');
if ((!strcmp(input, "q")) || (!strcmp(input, "Q")))


exit(0);
msg->setText (input);
tp->publish (topic, msg, 1, 0, 0);
delete msg;


}
}
catch (JMSException* jex)
{
```

```
jex->printStackTrace();
delete jex;
}


#ifdef _WIN32
return 1L;
#endif
};


class MtPubSub
{
InitialContext* m_ic;


public:
void runSample ()
{


Hashtable* table = new Hashtable();


table->put (SECURITY_PRINCIPAL, "anonymous");
table->put (SECURITY_CREDENTIALS, "anonymous");
table->put (PROVIDER_URL, "http://localhost:1856");
table->put (INITIAL_CONTEXT_FACTORY,


"fiorano.jms.runtime.naming.FioranoInitialContextFactory");
m_ic = new InitialContext (table);


MyRunnable runnable = (MyRunnable)((void*)newthread);
unsigned int i;


#ifdef _WIN32
_beginthreadex(NULL,0,runnable,m_ic,0,&i);


#else


pthread_create( (pthread_t*)&i, NULL, runnable, (void*)m_ic);
#endif
TopicConnectionFactory* tcf =


(TopicConnectionFactory*) m_ic->lookup ("primaryTCF");
Topic* topic =
(Topic*) m_ic->lookup ("primarytopic");
```

```
TopicConnection* tc = tcf->createTopicConnection ();
tc->start ();
TopicSession* ts = tc->createTopicSession (0, 1);
TopicSubscriber* tsub = ts->createSubscriber (topic, NULL, false);
tsub->setMessageListener (new MyListener ());
#ifdef _WIN32
Sleep(INFINITE);


#else


sleep(INFINITE);
#endif
}


~MtPubSub ()
{
if (m_ic != NULL)


delete (m_ic);
}
};


int main (char** args)
{
try
{


MtPubSub Pubsub;
Pubsub.runSample ();
}
catch (JMSException* jex)
{


jex->printStackTrace();


delete jex;
}
return 1;
}
```

Note: The same code is applicable for Multithreaded Clients on Solaris Platform. The above code can be used by running the sample application, MtPubSub, located in the <FMQ_DIR>\cpp\jni\samples\pubsub\mt directory.

Using Administration Application Programming Interface

Using C++ XA AdminAPIs

Admin Object Lookup

The lookup() operations on administered object such as connection factories can be achieved using the APIs provided by InitialContext. The following code snippet illustrates the usage of InitialContext to lookup administered objects.

```
env = new Hashtable ();
env->put (SECURITY_PRINCIPAL, "anonymous");
env->put (SECURITY_CREDENTIALS, "anonymous");
env->put (PROVIDER_URL,"http://localhost:1856");
env->put (INITIAL_CONTEXT_FACTORY,"fiorano.jms.runtime.naming.Fio-
ranoInitialContextFactory");


ic = new InitialContext (env);
cout <<"Created InitialContext... "<<endl;


acf = (MQAdminConnectionFactory *) ic->lookup ("primaryACF");


ac = acf->createMQAdminConnection ("admin", "passwd");
cout <<"Created Admin Connection..." <<endl;


adminService = ac->getMQAdminService();
cout <<"Received handle to Admin services... "<<endl;
```

Creating Admin objects and connection

The following code sample illustrates the process of creating admin objects such as queues, topic, XAConnection Factories etc.

```
try
{
metadata1 = new QueueMetaData ();
metadata1->setName ("RDBMSQueue1");
metadata1->setStorageType(1);
adminService->createQueue (metadata1);
cout <<"-->Successfully created XA enabled Queue :: " <<


metadata1->getName ()<<endl;
cout<<""<<endl;
 }
```

```
catch (JMSException* e){
e->printStackTrace();
 }


try
 {
metadata3 = new TopicMetaData ();
metadata3->setName ("RDBMSTopic1");
metadata3->setStorageType(1);


adminService->createTopic (metadata3);
cout << "-->Successfully created XA enabled Topic:: " <<

metadata3->getName ()<<endl;
cout <<""<<endl;
}


catch (JMSException* e)
{
e->printStackTrace();
}


try
{
ConnectionFactoryMetaData Name


xafactoryMetaData1 = new XAQueueConnectionFactoryMetaData();
xafactoryMetaData1->setName("MyXAQueueConnectionFactory1");
xafactoryMetaData1->setDescription ("XAQueue Connection


Factory1");
xafactoryMetaData1->setConnectURL ("http://localhost:1856");
adminService->createXAQueueConnectionFactory


(xafactoryMetaData1);
cout <<"Successfully created XAQueueConnectionFactory::" <<endl;
cout <<""<<endl;


}


catch (JMSException* e)
{
e->printStackTrace();
}
```

```
try


{
xafactoryMetaData3 = new XATopicConnectionFactoryMetaData();
xafactoryMetaData3->setName("MyXATopicConnectionFactory1");
xafactoryMetaData3->setDescription ("XATopic Connection


Factory1");
xafactoryMetaData3->setConnectURL ("http://localhost:1856");
adminService->createXATopicConnectionFactory


(xafactoryMetaData3);
cout <<"Successfully created XATopicConnectionFactory::" <<
xafactoryMetaData3<<endl;
cout <<""<<endl;
}


catch (JMSException* e)
{


e->printStackTrace();
}
try
{
xafactoryMetaData5 = new UnifiedXAConnectionFactoryMetaData();


 xafactoryMetaData5->setName("MyXAUnifiedConnectionFactory1");
 xafactoryMetaData5->setDescription ("XA Unified Connection


Factory1");
 xafactoryMetaData5->setConnectURL ("http://localhost:1856");
 adminService->createXAConnectionFactory (xafactoryMetaData5);
 cout <<"Successfully created XAConnectionFactory ::" <<endl;


 cout <<""<<endl;


}
catch (JMSException* e){
 e->printStackTrace();


}
```

Lookup Objects and XAConnection factories

The lookup operations on administered objects such as topic, queues, XAconnection factories etc can be achieved using the API provided by Initial Context. The following code snippet illustrates the usage of Initial Context to lookup administered objects.

```
cout <<"**************LOOKUP PROCESS
STARTED*****************"<<endl;
cout <<"\n --> Looking up XA enabled-Queues/Topics that were cre-
ated" <<endl;
try
{

 queue1 = (Queue *) ic->lookup ("RDBMSQueue1");
 cout <<"Looked up the XA enabled queue :: " << queue1->getQueue-

Name ()<<endl;
 queue2 = (Queue *) ic->lookup ("RDBMSQueue2");
 cout <<"Looked up the XA enabled queue :: " << queue2->getQueue

Name ()<<endl;
topic1 = (Topic *)ic->lookup("RDBMSTopic1");
cout <<"Looked up the XA enabled Topic :: "<< topic1->getTopic

Name()<<endl;
topic2 = (Topic *)ic->lookup("RDBMSTopic2");
cout <<"Looked up the XA enabled Topic ::"<< topic2->getTopic

Name()<<endl;
cout <<""<<endl;
}

catch (JMSException* e)
{
e->printStackTrace();
}

cout <<"\n --> Looking up created Connection Factories  "<<endl;
try
{

 QueueConnectionFactory1 = (XAQueueConnectionFactory *) ic->lookup

("MyXAQueueConnectionFactory1");
cout <<"Looked up XAQueueConnectionFactory : " << endl;
QueueConnectionFactory2 = (XAQueueConnectionFactory *) ic->lookup

("MyXAQueueConnectionFactory2");
cout <<"Looked up XAQueueConnectionFactory : " << endl;
cout <<""<<endl;
topicConnectionFactory1 = (XATopicConnectionFactory *) ic->lookup
```

```
("MyXATopicConnectionFactory1");
cout <<"Looked up XATopicConnectionFactory : " << endl;
topicConnectionFactory2 = (XATopicConnectionFactory *) ic->lookup

("MyXATopicConnectionFactory2");
cout <<"Looked up XATopicConnectionFactory : " << endl;
cout <<""<<endl;

unifiedConnectionFactory1 = (XAConnectionFactory *) ic->lookup

("MyXAUnifiedConnectionFactory1");
cout <<"Looked up XAConnectionFactory : " << endl;
unifiedConnectionFactory2 = (XAConnectionFactory *) ic->lookup

("MyXAUnifiedConnectionFactory2");
cout <<"Looked up XAConnectionFactory : " << endl;
ac->close();

}

catch (JMSException* e)
{
e->printStackTrace();
}
```

The above code can be used by running the sample application, AdminTestXA, located in the <FMQ_DIR>\cpp\jni\samples\pubsub\XA\Admin directory.

Using Local Transaction APIs

Admin Object Lookup

The lookup() operations on administered object such as queue, topic, connection factories etc can be achieved using the APIs provided by InitialContext. The following code snippet illustrates the usage of InitialContext to lookup administered objects.

```
try
{
    env->put(SECURITY_PRINCIPAL, "anonymous");
    env->put(SECURITY_CREDENTIALS, "anonymous");
    env->put(PROVIDER_URL, "http://localhost:1856");
    env->put(INITIAL_CONTEXT_FACTORY,
    "fiorano.jms.runtime.naming.FioranoInitialContextFactory");
    ic = new InitialContext(env);
    cout << "Initial context created..." << endl;

tory*) ic->lookup("primaryXAQCF");
   XATopicConnectionFactory *xatcf = (XATopicConnectionFactory*)
```

```
ic->lookup("primaryXATCF");
  cout << "Looked up Connection Factories..." << endl;
 // 1.2  Lookup Topics/Queues
  // Queue *xaqueue = (Queue*) ic->lookup("primaryRDBMSQueue");
  Topic *xatopic = (Topic*) ic->lookup("primaryRDBMSTopic");
  cout << "Looked up Destinations..." << endl;
```

Creating XA connection and setting id for local transaction

The following code illustrates the process of creating a connection over the connection factories, setting client ids over the connection and further creating session over the connection. These objects enables JMS provider to perform messaging for local transaction.

```
XAQueueConnection *xaqConn = xaqcf->createXAQueueConnection();
XATopicConnection *xatConn = xatcf->createXATopicConnection();
cout << "Created XA Connection..." << endl;

xaqConn->setClientID("Client_1");
xatConn->setClientID("Client_2");

xaqConn->start();
xatConn->start();

XAQueueSession *xaqs = xaqConn->createXAQueueSession();
XATopicSession *xats = xatConn->createXATopicSession();
cout << "Created XA Session..." << endl;

QueueSender *queueSender = (xaqs->getQueueSession())->create-
Sender(xaqueue);
TopicPublisher *publisher = (xats->getTopicSession())->createPub-
lisher(xatopic);
cout << "Sender/Publisher created..." << endl;
Sending/publishing message in local transaction
```

The following code illustrates the process of sending message through local transaction and further it illustrates committing local transaction using commit() method of QASession API.

```
 TextMessage *txtMsg = xaqs->createTextMessage();
TextMessage *txtMsg1 = xats->createTextMessage();
for (int i = 0; i < 10; i++) {
char *text = new char[100];
sprintf(text, "Local Transaction: %d", i);
txtMsg->setText(text);
queueSender->send(txtMsg);
cout << "Sent text: " << txtMsg->getText() << endl;
delete text;
}
for (i = 0; i < 10; i++) {
char *text = new char[100];
sprintf(text, "Local Transaction: %d", i);
txtMsg1->setText(text);
publisher->publish(txtMsg1);
cout << "Published text: " << txtMsg1->getText() << endl;
delete text;
}
xaqs->getQueueSession())->commit();
xats->getTopicSession())->commit();
cout << "Local Transaction committed..." << endl;
Getting XA resources for distributed transaction
```

The following code illustrates the process of creating ids and getting resources for distributed transactions. These resources enables JMS provider to perform messaging through distributed transaction.

```
XAResource *xaResource = xaqs->getXAResource();
XAResource *xaResource1 = xats->getXAResource();
cout << "Got XA Resource..." << endl;
char *branchId = "BranchID11";
char *gtid = "globalTransactionId1";
Xid *xid = new FioranoXid(gtid, branchId, 0);
char *branchId1 = "BranchID2";

char *gtid1 = "globalTransactionId2";
Xid *xid1 = new FioranoXid(gtid1, branchId1, 0);
cout << "Xids created..." << endl;

xaResource->start(xid, XAResource::TMNOFLAGS);
xaResource1->start(xid1, XAResource::TMNOFLAGS);
cout << "Resource started..." << endl;
```

Publishing the message for Distributed (global) Transaction

The following code illustrates the process of sending message in distributed transaction, after completion of message sending process; code illustrates ending the XAresources using end method of XA resources API. Further it commit the transaction using commit() method of XAResource API. At the end all connection is closed.

```
for (i = 0; i < 10; i++) {
char *text = new char[100];
sprintf(text, "Global Transaction: %d", i);
txtMsg->setText(text);
queueSender->send(txtMsg);
cout << "Sent text: " << txtMsg->getText() << endl;
delete text;


}


 for (i = 0; i < 10; i++)


{
char *text = new char[100];
sprintf(text, "Global Transaction: %d", i);
txtMsg1->setText(text);
publisher->publish(txtMsg1);
cout << "Published text: " << txtMsg1->getText() << endl;
delete text;}


xaResource->end(xid, XAResource::TMSUCCESS);
xaResource1->end(xid1, XAResource::TMSUCCESS);
cout << "Ended the transaction..." << endl;


int flag = xaResource->prepare(xid);
int flag1 = xaResource1->prepare(xid1);
cout << "Prepared the transaction..." << endl;


if (flag1 == XAResource::XA_OK) {
xaResource1->commit(xid1, false);
}


if (flag == XAResource::XA_OK)
{
xaResource->commit(xid, false);
}
```

```
cout << "Committed the transaction..." << endl;
cout << "Closing sessions and connections..." << endl;
xaqs->close();
xats->close();


xaqConn->close();
xatConn->close();
```

The above code can be used by running the sample application, Local_Transaction, located in the <FMQ_DIR>\cpp\jni\samples\pubsub\XA\Local_Transaction directory.

Using C++ Samples

Organization of Samples Provided

The C++RTL samples provided in the FioranoMQ client installation are organized into the following directories that can be found in the cpp\jni\samples directory of the installation.

ptp

- basic Contains sample programs illustrating the JMS PTP API
- reqrep Contains sample programs illustrating the use of Request-Reply mechanism
- HTTP Contains sample programs illustrating the use of HTTP protocol for basic JMS ptp functionality

pubsub

- **basic** Contains sample programs illustrating the JMS PTP API
- **mt** Contains sample programs illustrating the multi-threaded functionality
- **objmsg** Contains sample programs illustrating the object message functionality
- **reqrep** Contains sample programs illustrating the use of Request-Reply mechanism
- **HTTP** Contains sample programs illustrating the use of HTTP protocol for basic JMS pubsub functionality

Unified Domain

- **CreateCF** Contains sample programs illustrating the use of FioranoMQ Administration APIs for creation of UnifiedConnectionFactories.
- **Rollback** Contains sample programs illustrating the basic rollback operations in consumer side
- **SendReceive** Contains sample programs illustrating the basic Producer-Consumer operations
- **Transacted** Contains sample programs illustrating the basic Producer-Con-sumer operations in Transacted session

XA

- **Admin** Contains sample programs illustrating the use of the FioranoMQ Administration APIs for creation of JMS Administered object**s**
- **XASamples** Contains sample programs illustrating the XA capabilities of FioranoMQ.

Compiling and Running the Samples

On Windows

Compiling the Samples

Before running these samples, you need to perform the following:

1. Set JAVA_HOME=:\j2sdk1.4.1\jre
2. Set FMQ_DIR= \FioranoInstaller2\fmq

   In case you has a client only installer he can set the environment variable CLASSPATH and get working
3. Set CLASSPATH=. \FioranoInstaller2\fmq\lib\client\fmq-cli-ent.jar; \FioranoInstaller2\fmq\clients\cpp\jni\msgAdapter\java\lib\fmq-cpp-client-msg-adapter.jar

Other than the classpath and environment settings, you also need to have the following in the system path.

- fmq-cpp-client-msg-adapter.dll
- fmq-jni-cpprtl.dll

The samples provided with C++RTL can be compiled using the script cppclient-build.bat, provided along with the C++ runtime package. The following variables in the script file should be set to the appropriate paths:

- VC_DIR The root directory of the Microsoft Visual C++ installation
- FMQ_DIR The root directory of FioranoMQ installation.

The following example illustrates how to use the cppclientbuild.bat script for compiling a sample application.

cppclientbuild.bat Sample.c

The libraries required for compiling the CRTL samples (apart from the default VC libraries) are:

- fmq-jni-cpprtl.lib
- jvm.lib.

Using VC++ for compiling and running the samples

- The Code Generation option needs to be set to Multi threaded DLL
- Include directory needs to point to cpp\jni\include folder
- The program needs to be linked to fmq-jni-cpprtl.lib and jvm.lib.

Running the Samples

After compiling the samples as explained above, they can be executed by using the corresponding executable file generated.

Sample [<arg1> <arg2> ...]
fmq-msg-adapter.dll should be present in System path before running the executables.

On Solaris

Compiling the Samples

The samples provided with C++ Runtime can be compiled using the script cppcli-entbuild.sh that comes with the installation. This script file uses the GCC compiler and assumes that the same is available on the concerned Solaris machine.

The following example illustrates how to compile a sample using cppclient-build.sh.

./**cppclientbuild.sh Sample.**c
After compilation a bin file (Sample.bin for example above) is created.

Running the Samples

The bin file formed after compilation can be executed from the console as follows.

Sample.bin [<arg1> <arg2> ...]

Chapter 3: Trouble Shooting C++RTL Support for ObjectMessages

The latest release of C++RTL has added support for JMS ObjectMessages. This allows the user to send and receive Java objects across C++ and Java applications and also across just C++ applications. However for this purpose the user would have to directly deal with JNI APIs.

The sample application below is a C++ Subscriber that received the Java Integer object and prints its integer value using JNI APIs. This is an asynchronous subscriber that un-wraps the Java Integer object in its onMessage method. The publisher application in this case could be either a C++/Java application.

```
#include
#include
#include
#include


class MyListener : public MessageListener
{
public:


void onMessage (Message* msg)


{


 try


 {


 ObjectMessage* tMsg = (ObjectMessage*)msg;


 jobject str = tMsg->getObject ();


 JNIEnv* localEnv = m_jvm.getTLSenv();


   jclass m_cls = localEnv->FindClass("java/lang/
Integer");
 jmethodID getint = localEnv->GetMethodID(m_cls,
"intValue", "()I");
 jint inte = localEnv->CallIntMethod(str, get-
int);
printf("Java Object Integer Value :: %d ",inte);
```

```
 }
    catch (JMSException* ex)

{
 printf("Error reading message ");
 ex->printStackTrace();

 }

}

};

class MyEListener : public ExceptionListener
{
public:

MyEListener () : ExceptionListener ()
{
}

void onException (JMSException* je)
{
printf ("Got Exception ");
    je->printStackTrace ();

}

};

class Subscriber
{

   InitialContext*  m_ic;

   public:

void runTest ()
{
// Create the Fiorano Initial Context Object
 //
```

```
    Hashtable* table = new Hashtable();


    table->put ("java.naming.provider.url", "http://
localhost:1856");
    table->put ("java.naming.factory.initial",
                   "fiorano.jms.runtime.naming.FioranoInitialContex-
tFactory");
    m_ic = new InitialContext (table);


    // create the topic connection factory
    TopicConnectionFactory* tcf =
      (TopicConnectionFactory*) m_ic->lookup
("primaryTCF");
    Topic* topic = (Topic*) m_ic->lookup ("primary-
topic");


    // create the topic connection
    cout << "Creating topic connection" << endl;


  TopicConnection* tc = tcf->createTopicConnection ();
tc->start ();


ExceptionListener* exListener = new MyEListener ();
tc->setExceptionListener (exListener);


// create the topic session
cout << "Creating topic session" << endl;


TopicSession* ts = tc->createTopicSession (0, 1);


        // create the topic subcriber and set the message listener
cout << "Creating topic subscriber on primary Topic" << endl;
  TopicSubscriber* tsub = ts->createSubscriber (topic, NULL,
false);
tsub->setMessageListener (new MyListener ());


// Wait for program to terminate.
int dummy;
scanf("%d",&dummy);


tc->stop ();


 }
```

```
    ~Subscriber ()
 {
if (m_ic != NULL)
    delete (m_ic);
 }


};


/**


 * Implements an asynchronous listener to listen


 * for messages published on the topic - "primaryTopic".




 */
__cdecl main (char** args)
{


 try


 {
Subscriber sub;
sub.runTest ();


 }
    catch (JMSException* jex)
 {


printf("Error while executing test case ");
jex->printStackTrace();
 }


 return 1;
}
```

Using the C++ ObjectMessage support user defined types (UDT) could also be sent across. The user should set USR_DIR environment variable to point to the directory where his UDT lies. The C++RTL uses this directory setting (if set) to load the UDT.

The following sample shows how a Publisher/Subscriber deals with a UDT (Employee class).

```
public class Employee implements java.io.Serializable
{
private String m_strName;
private int  m_age;
private double m_salary;
public Employee(String name, int age, double salary)
{
    m_strName = name;
    m_age = age;
    m_salary = salary;
}
public int getAge()
{
    return m_age;
}
public String getName()
{
    return m_strName;
}
public double getSalary()
{
    return m_salary;
}
}
```

A sample publisher dealing with the employee UDT is as follows:

```
// Object message related code :: JNI
    JNIEnv* localEnv = m_jvm.getTLSenv();


    jclass m_cls = localEnv->FindClass("Employee");
    jmethodID constr = localEnv->GetMethodID
(m_cls, "", "(Ljava/lang/String;ID)V");


    //Get inputs for Employee


    printf("Enter data for Employee :: ");


    char name[128];
```

```
    int age;
    double salary;
    printf("Name   :: ");
    scanf("%s", name);
    printf("Age    :: ");
    scanf("%d", &age);
    printf("Salary :: ");
    scanf("%lf", &salary);
jstring jname = localEnv->NewStringUTF(name);
jobject      object = localEnv->NewObject(m_cls, constr,
jname, age, salary);


//Object message related code :: JNI
ObjectMessage* mesg = ts->createObjectMessage(object);
tpub->publish(mesg);
```

The onMessage of a Subscriber dealing with the Employee UDT is shown below.

```
void onMessage (Message* msg)
{
 try


 {


  ObjectMessage* tMsg = (ObjectMessage*)msg;


  jobject str = tMsg->getObject ();


  // Object message related code :: JNI
  JNIEnv* localEnv = m_jvm.getTLSenv();
  jclass m_cls = localEnv->FindClass("Employee");


jmethodID getName = localEnv->GetMethodID(m_cls,
"getName",
"() Ljava/lang/String;");


  jmethodID getAge = localEnv->GetMethodID(m_cls,
"getAge", "()I");


jmethodID getSalary = localEnv->GetMethodID(m_cls,
"getSalary", "()D");


  jobject udtObj = tMsg->getObject();
```

```
  jstring  jstrName = (jstring)localEnv->CallObject-
Method(udtObj, getName);


  jint jage = localEnv->CallIntMethod(udtObj,
getAge);


  jdouble  jsalary  = localEnv->CallDou-
bleMethod(udtObj, getSalary);


  jboolean b = false;
  const char* name = localEnv->GetStringUTF-
Chars(jstrName, &b);
  int length = localEnv->GetStringUTFLength(jstr-
Name);
  if(length == 0)
       name = NULL;


  char* nameStr = new char[length + 1];
  strncpy(nameStr,name,length);
  nameStr[length] = '\0';


  localEnv->ReleaseStringUTFChars(jstrName,name);


  // Object message related code :: JNI


      printf("Java Object UDT data ");
      printf("Class Employee :: ");
      printf("Name  :: %s ",nameStr);
      printf("Age  :: %d ",jage);
      printf("Salary :: %f ",jsalary);


 }
 catch (JMSException* ex)
 {


      printf("Error reading message ");
      ex->printStackTrace();
 }
}
```

For accessing the JNIEnv to deal with ObjectMessages the user would just have to use the following API (See the above code snippets for usage).

```
JNIEnv* localEnv = m_jvm.getTLSenv();
```

Here m_jvm is an C++ object instance accessible to the user and the API get-TLSenv() is to be used to access the environment pointer of the current thread of execution.

Deprecated APIs: 4.6 and Below

Although some of the APIs have been changed, care has been taken to provide full backward compatibility.

The older version of the InitialContext (FioranoInitialContext) could still be used in the same way as before.The code snippet below shows the usage of FioranoInitialContext to lookup administered objects.

```
FioranoInitialContext* m_ic;
TopicConnectionFactory* tcf;
TopicConnection* tc;
Topic* topic;

// creating fioranoinitial context object
m_ic = new FioranoInitialContext ();
//bind the fioranoinitial
context m_ic->bind ("localhost", 1856);

// create the topic connection factory
tcf = (TopicConnectionFactory*) m_ic->lookup("primaryTCF");
topic = (Topic*) m_ic->lookup ("primarytopic");

// Dispose the intial context after the lookup is done
m_ic->dispose ();
```

Compilation Issues

When applications based on the C++RTL are compiled with rpc.h or corba.h or with include files in older versions of Visual C++, clients face problems due to clashes in the definition of the datatype byte.

Clients who are using either of these should use the datatype fmqbyte in all APIs of the C++RTL in place of byte.