software AG

webMethods EntireX

Reliable RPC

Version 9.5 SP1

November 2013

webMethods EntireX

Table of Contents

Reliable RPC

Reliable RPC is the EntireX implementation of a reliable messaging system. It combines EntireX RPC technology and persistence, which is implemented with units of work (UOWs).

Introduction to Reliable RPC	Concepts of Reliable RPC; Broker configuration for Reliable RPC.
Reliable RPC for C Wrapper	Reliable RPC for C Wrapper. Also includes an `AUTO_COMMIT` call sequence example.
Reliable RPC for COBOL Wrapper	Reliable RPC for COBOL Wrapper.
Reliable RPC for DCOM Wrapper	Reliable RPC for DCOM Wrapper.
Reliable RPC for .NET Wrapper	Reliable RPC for .NET Wrapper.
Reliable RPC for Java Wrapper	Reliable RPC for Java Wrapper.
Reliable RPC for XML/SOAP Wrapper	Reliable RPC for XML/SOAP Wrapper.

1 Overview of Reliable RPC

Introduction to Reliable RPC

In the architecture of modern e-business applications (such as SOA), loosely coupled systems are becoming more and more important. Reliable messaging is one important technology for this type of system.

Reliable RPC is the EntireX implementation of a reliable messaging system. It combines EntireX RPC technology and persistence, which is implemented with units of work (UOWs).

- Reliable RPC allows asynchronous calls ("fire and forget")
- Reliable RPC is supported by most EntireX wrappers
- Reliable RPC messages are stored in the Broker's persistent store until a server is available
- Reliable RPC clients are able to request the status of the messages they have sent
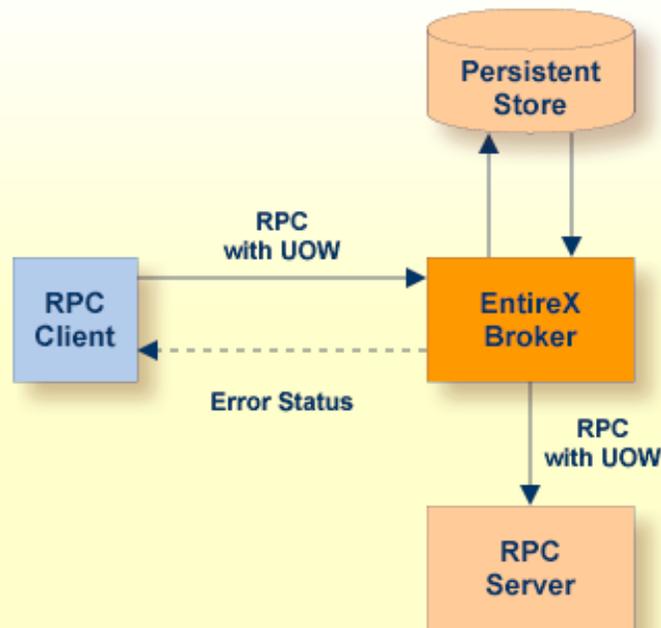
Reliable RPC is used to send messages to a persisted Broker service. The messages are described by an IDL program that contains only `IN` parameters. The client interface object and the server interface object are generated from this IDL file, using the respective Software AG component.

Reliable RPC is enabled at runtime. The client has to set one of two different modes before issuing a reliable RPC request:

- `AUTO_COMMIT`
- `CLIENT_COMMIT`

While `AUTO_COMMIT` commits each RPC message implicitly after sending it, a series of RPC messages sent in a unit of work (UOW) can be committed or rolled back explicitly using `CLIENT_COMMIT` mode.

The server is implemented and configured in the same way as for normal RPC.

Broker Configuration

A Broker configuration with `PSTORE` is recommended. This enables the Broker to store the messages for more than one Broker session. These messages are still available after Broker restart. The attributes `STORE`, `PSTORE`, and `PSTORE-TYPE` in the Broker attribute file can be used to configure this feature. The lifetime of the messages and the status information can be configured with the attributes `UWTIME` and `UWSTAT-LIFETIME`. Other attributes such as `MAX-MESSAGES-IN-UOW`, `MAX-UOWS` and `MAX-UOW-MESSAGE-LENGTH` may be used in addition to configure the units of work. See *Broker Attributes* in the administration documentation.

The following rules apply:

Java

The result of the method `RPCService.getStatusOfMessage` depends on the configuration of the unit of work status lifetime. If the status is not stored longer than the message, the method returns `(not available)`.

C

The result of the procedure *ERXGetReliableStatus* depends on the configuration of the unit of work status lifetime in the EntireX Broker configuration. If the status is not stored longer than the message, the procedure returns the error code 00780305 (`no matching UOW found`).

COBOL

The result of the generic RPC function call "RS" - get reliable status depends on the configuration of the unit of work status lifetime in the EntireX Broker configuration. See `COMM-FUNCTION` in the COBOL Wrapper documentation. If the status is not stored longer than the message, the function call returns the error code 00780305 (`no matching UOW found`).

.NET

The result of the function `Service.GetReliableStatus` depends on the configuration of the unit of work status lifetime in the EntireX Broker configuration. If the status is not stored longer than the message, the function returns the error code 00780305 (`no matching UOW found`).

DCOM

The result of the function RPCService.get_StatusOfMessage depends on the configuration of the unit of work status lifetime in the EntireX Broker configuration. If the status is not stored longer than the message, the function returns the error code 00780305 (`no matching UOW found`).

2 Reliable RPC for C Wrapper

Writing a Client

This section shows a reliable RPC client for `CLIENT_COMMIT` mode. All methods for reliable RPC are defined in *erx.h*. The methods applicable to reliable RPC as described under *API Function Descriptions for Reliable RPC* are:

- `ERXGetReliableState`
- `ERXSetReliableState`
- `ERXReliableCommit`
- `ERXReliableRollback`
- `ERXGetReliableID`
- `ERXGetReliableStatus`

The example below is included as source in directory *examples/ReliableRPC/CClient*.

Step 1: Base Declarations Required by the C Wrapper

Step 1a: Include the Generated Header File

Define the generated client header file. This header file includes the RPC runtime header file erx.h and defines structures and prototypes for your RPC messages.

```
/* include generated header file */
#include "cmail.h"
```

Step 1b: Define Global Variables to Communicate with the Client Interface Objects

```
/* Required global variables for the CLIENT interface */
ERXeReturnCode             ERXrc;
ERX_CLIENT_IDENTIFICATION  ERXClient;
ERX_SERVER_ADDRESS         ERXServer;
ERX_SERVER_ADDRESS         ERXServerDefault;
ERXCallId                  ERXCallID;
ERX_ERROR_INFO             ERXErrorInfo;
```

Step 2: Required Settings for the C Wrapper

Step 2a: Identify the User with a Broker User ID

For implicit broker logon, if required in your environment, the client password can be given here. It is provided then through the RPC interface object call.

```
/* set client identification */
memset( &ERXClient, 0, sizeof(ERXClient) );
strcpy( (char*) ERXClient.szUserId, "ERX-USER" );
strcpy( (char*) ERXClient.szPassword, "ERX_PASS");
```

Step 2b: Set the Broker and Service to be Called

Your application will wait a maximum of 55 seconds for a server response. If the server does not answer within this period, the broker gives your program control again with an error code 00740074.

```
ERXServer.Medium = ERX_TM_BROKER_LIBRARY;
ERXServer.ulTimeOut = 55;

/* set Broker-Id, server-name, class-name and service-name */
strcpy( (char*) ERXServer.Address.BROKER.szEtbidName,   "ETB001" );
strcpy( (char*) ERXServer.Address.BROKER.szServerName,  "SRV1" );
strcpy( (char*) ERXServer.Address.BROKER.szClassName,   "RPC" );
strcpy( (char*) ERXServer.Address.BROKER.szServiceName, "CALLNAT" );
```

Step 3: Register with the RPC Runtime

As a general rule, you have to register the RPC runtime before you use it. After registration, the RPC runtime holds information on a per-thread basis. See also *Using the RPC Runtime* under *Writing Advanced Applications with the C Wrapper*.

```
/* register to the RPC runtime */
ERXrc = ERXRegister(ERX_CURRENT_VERSION );
If ( ERX_FAILED( ERXrc ) )
{
/* code for  error handling */
}
```

Step 4: Broker Logon

We logon by EntireX Broker.

```
/* Logon to EntireX Broker Middleware */
ERXrc = ERXLogon( &ERXClient,
                   ERXServer.Address.BROKER_Library.szEtbidName );
if(ERX_FAILED(ERXrc))
{
/* code for  error handling */
}
```

Step 5: Set Reliable-State

Before reliable RPC can be used, the reliable state must be set to either `ERX_RELIABLE_CLIENT_COMMIT` or `ERX_RELIABLE_AUTO_COMMIT`.

```
/* Set reliable RPC state to client commit */
ERXrc = ERXSetReliableState(ERX_RELIABLE_CLIENT_COMMIT);
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 6: Send the RPC Message

The RPC interface object `SENDMAIL` is called as a C procedure. See *Calling Servers as Procedures or Functions* under *Software AG IDL to C Mapping* in the C Wrapper documentation.

```
/* do the remote procedure call */
SENDMAIL( gTo, gSubject, gText);
```

Step 7: Get the Reliable RPC Message ID

Get the reliable RPC message ID before you commit any reliable RPC messages, otherwise the reliable ID will be lost and checking for the RPC message status will not be possible.

```
/* Get the reliable ID */
ERXrc = ERXGetReliableID( &ERXServer, pReliableID );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 8: Check the Reliable RPC Message Status

After the reliable RPC message ID has been got, you can query the status of the reliable RPC message. This is a separate call independent of any reliable RPC messages, so we use the default server connection (`ERXServerDefault`). Valid reliable RPC message states can be found in header file *etbcdef.h*. See *Broker ACI Control Block Definition* in the ACI for C documentation.

See *Using Persistence and Units of Work* in the general administration documentation, *Understanding UOW Status* under *Using Persistence and Units of Work* in the general administration documentation and *Broker UOW Status Transition* under *Concepts of Persistent Messaging* in the general administration documentation for more information.

```
/* Check the reliable RPC message status */
ERXrc = ERXGetReliableStatus( &ERXClient,
                              &ERXServerDefault,
                              pReliableID,
                              pReliableStatus );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 9: Send a Second RPC message

Send a second reliable RPC message.

```
/* do the remote procedure call */
SENDMAIL( gTo, gSubject, gText);
```

Step 10: Commit Both Reliable RPC Messages

Now we commit both reliable RPC messages. This will deliver all reliable RPC messages to the server if it is available.

```
/* Commit all made reliable RPC messages */
ERXrc = ERXReliableCommit( &ERXServer );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 11: Reset ERX_SERVER_ADDRESS

For reliable RPC, the ERX_SERVER_ADDRESS will be overwritten by the RPC runtime, so it is necessary to reset the ERX_SERVER_ADDRESS structure with the required values.

```
/*
 * After a ERXReliableCommit we have to use a new server connection
 * so we restore our default server connection for further calls.
 */
memcpy(&ERXServer, &ERXServerDefault, sizeof(ERX_SERVER_ADDRESS));
```

Step 12: Check the Reliable RPC Message Status

To determine that reliable RPC messages are delivered, we query the reliable RPC message status again. See also *Step 8* above.

Step 13: Send a Third RPC message

Send a third reliable RPC message.

```
/* do the remote procedure call */
SENDMAIL( gTo, gSubject, gText);
```

Step 14: Get the Reliable RPC Message ID

Get the reliable RPC message ID. See also *Step 7*.

```
/* Get the reliable ID */
ERXrc = ERXGetReliableID( &ERXServer, pReliableID );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 15: Check the Reliable RPC Message Status

After the reliable RPC message ID has been got, query the status of the reliable RPC message again.

```
/* Check the reliable RPC message status */
ERXrc = ERXGetReliableStatus( &ERXClient,
                              &ERXServerDefault,
                              pReliableID,
                              pReliableStatus );
if( ERX_FAILED(ERXrc) )
{
```

```
/* code for  error handling */
}
```

Step 16: Roll back the Third Message

Roll back the current reliable RPC message.

```
/* Roll back Message 3 */
ERXrc = ERXReliableRollback( &ERXServer );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 17: Check the Reliable RPC Message Status

After rolling back the reliable RPC message, query the status of the reliable RPC message.

```
/* Get the reliable RPC message status */
ERXrc = ERXGetReliableStatus( &ERXClient,
                              &ERXServerDefault,
                              pReliableID,
                              pReliableStatus );
if( ERX_FAILED(ERXrc) )
{
/* code for  error handling */
}
```

Step 18: Broker Logoff

Log off from EntireX Broker.

```
/* Logoff from EntireX Broker Middleware */
ERXrc = ERXLogoff( &ERXClient,
                   ERXServerDefault.Address.BROKER_Library.szEtbidName );
if ( ERX_FAILED( ERXrc ) )
{
/* code for  error handling */
}
```

Step 19: Deregister with the RPC Runtime

As a general rule, after using the RPC runtime you should unregister from it. This will free all resources held by the RPC runtime for the caller. See *Using the RPC Runtime* under *Writing Advanced Applications with the C Wrapper* for more information.

```
/* unregister to the RPC runtime */
ERXUnregister();
```

Writing a Client using AUTO COMMIT

This section gives some hints for reliable RPC AUTO_COMMIT mode. It is not a complete example and shows only the correct order of reliable RPC method calls. The reliable ID to check the message status must be retrieved immediately after the reliable RPC message is sent and before any other RPC runtime calls - otherwise the reliable ID is lost and retrieving the message status is not possible.

```
/* Initialize pERXServer */
...

/*
 * After initializing pERXServer  with your connection settings (broker ID,
 * server-name, calss-name, service-name) create a copy of it
 * (pERXDefaultServer). Use this copy to resolve the reliable status after
 * a reliable RPC message.
 */
memcpy(pERXServer, pERXDefaultServer, sizeof(ERX_SERVER_ADDRESS));

...

/* Set reliable state to AUTO_COMMIT */
ERXSetReliableState( ERX_RELIABLE_AUTO_COMMIT );

...



/* reliable RPC message 1 */
SENDMAIL( gTo, gSubject, gText );
/*
 * The reliable ID must be resolved directly
 * after a reliable RPC message
 */
ERXGetReliableID( pERXServer, pReliableID );



...
```

```
/* Resolve the reliable status */
ERXGetReliableStatus( pERXClient, pERXDefaultServer, pReliableID,
                      pReliableStatus );


...


/* For a second AUTO_COMMIT RPC message, use a new server connection */
memcpy(pERXServer, pERXDefaultServer, sizeof(ERX_SERVER_ADDRESS));


...


/* reliable RPC message 2 */
SENDMAIL( gTo, gSubject, gText );
/*
 * The reliable ID must be resolved directly
 * after a reliable RPC message
 */
ERXGetReliableID( pERXServer, pReliableID );



...


/* Resolve the reliable status */
ERXGetReliableStatus( pERXClient, pERXDefaultServer, pReliableID,
                      pReliableStatus );


...
```

Writing a Server

There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it returns an error code different to 0000000. This causes a cancel of the transaction (unit of work inside the Broker) and the error code is written to the user status field of the unit of work.

For writing reliable RPC servers, see *Using the C Wrapper for the Server Side (z/OS, UNIX, Windows, BS2000/OSD, IBM i).*

To execute a reliable RPC service with an RPC server, the parameter logon must be set to "YES", see

- *Configuring the BS2000/OSD Batch RPC Server* under *Administering the BS2000/OSD Batch RPC Server*

- *Setting Server Parameters for the RPC Server* in the UNIX and Windows administration documentation

3 Reliable RPC for COBOL Wrapper

Writing a Client

The following steps describe how to write a COBOL reliable RPC client program with the scenario *Using the COBOL Wrapper for CICS with DFHCOMMAREA Calling Convention (z/OS and z/VSE)* in the COBOL Wrapper documentation and Linkage access to RPC communication.

Reliable RPC requires an explicit broker logon. See *Using Broker Logon and Logoff* under *Writing Applications with the COBOL Wrapper*.

Step 1: Declare the Data Structures for RPC Client Interface Objects

For every program definition in the Software AG IDL file, the templates will generate a copybook file that describes the customer data of the interface as a COBOL structure. For ease of use, the copybook can be embedded into the RPC client program.

However, if more appropriate, customer data structures can be used. In this case the COBOL data types and structures must match the interfaces of the generated client interface objects, otherwise unpredictable results will occur.

```
* Declare the customer data of the generated RPC interface
 01 SENDMAIL.
    02 SM-COMA.
       03 SM-TOADDRESS          PIC  X(60).
       03 SM-SUBJECT            PIC  X(20).
       03 SM-TEXT               PIC X(100).
```

Step 2: Declare and Initialize the RPC Communication Area

The RPC communication area must be declared and initialized in your RPC client program as follows:

```
* Declare RPC communication area
 02 ERX-COMMUNICATION-AREA.
    COPY ERXCOMM.
 . . . . . .

* Initialize RPC communication area
 INITIALIZE ERX-COMMUNICATION-AREA.
 MOVE "2000"            to COMM-VERSION.
```

Step 3: Required Settings in the RPC Communication Area

The following settings to the RPC communication area are required as a minimum to use the COBOL Wrapper. These settings have to be applied in your RPC client program. It is not possible to generate any defaults into your client interface objects:

```
* assign the broker to talk with
 MOVE "localhost:1971" to COMM-ETB-BROKER-ID.

* assign the server to talk with
 MOVE "RPC"             to COMM-ETB-SERVER-CLASS.
 MOVE "SRV1"            to COMM-ETB-SERVER-NAME.
 MOVE "CALLNAT"         to COMM-ETB-SERVICE-NAME.
* assign the user ID for Broker logon
 MOVE "ERXUSER"         to COMM-USERID.
 MOVE "PASSWORD"        to COMM-PASSWORD.
```

Step 4a: Perform a Broker Logon

```
MOVE "LO" TO COMM-FUNCTION.
EXEC CICS LINK
  PROGRAM  ("COBSRVI")
  COMMAREA (ERX-COMMUNICATION-AREA)
  LENGTH   (LENGTH OF ERX-COMMUNICATION-AREA)
  RESP     (CICS-RESP1)
  RESP2    (CICS-RESP2)
END-EXEC.
```

Step 4b: Examine the Error Code

Check whether the logon call was successful or not.

Step 5: Enable Reliable RPC with CLIENT_COMMIT

Before reliable RPC can be used, the reliable state must be set to either `ERX_RELIABLE_CLIENT_COMMIT` or `ERX_RELIABLE_AUTO_COMMIT`.

- "C" - `CLIENT_COMMIT`
- "A" - `AUTO_COMMIT`

```
* Set the reliable RPC mode
 MOVE "C" TO COMM-RELIABLE-STATE.
```

Step 6a: Send the RPC Message

The RPC message is sent using the EXEC CICS LINK interface.

```
* Send the RPC message
 MOVE DFHRESP(NORMAL) TO CICS-RESP1.
 MOVE DFHRESP(NORMAL) TO CICS-RESP2.
 MOVE ZEROES          TO COMM-RETURN-CODE.
 EXEC CICS LINK
   PROGRAM  ("SENDMAIL")
   RESP     (CICS-RESP1)
   RESP2    (CICS-RESP2)
   COMMAREA (SENDMAIL)
   LENGTH   (LENGTH OF SENDMAIL)
 END-EXEC.
```

Step 6b: Examine the Error Code

When the RPC message is returned, it needs to be checked whether it was successful or not:

```
IF COMM-RETURN-CODE IS = ZERO
   Perform success-handling
ELSE
   Perform error-handling
END-IF.
```

The field COMM-RETURN-CODE in the RPC communication area contains the error provided by the COBOL Wrapper. For the error messages returned, see *Error Messages and Codes*.

> **Note:** After successful call (Step 6a) the UOWID is available in the RPC communication area field COMM-ETB-UOW-ID. See *The RPC Communication Area (Reference)* in the COBOL Wrapper documentation.

Step 7a: Check the Reliable RPC Message Status

To determine that reliable RPC messages are delivered, the reliable RPC message status can be queried. See *Understanding UOW Status* under *Using Persistence and Units of Work* in the general administration documentation and *Broker UOW Status Transition* under *Concepts of Persistent Messaging* in the general administration documentation for more information.

```
MOVE DFHRESP(NORMAL) TO CICS-RESP1.
MOVE DFHRESP(NORMAL) TO CICS-RESP2.
MOVE "RS" TO COMM-FUNCTION.
MOVE ZEROES TO COMM-RETURN-CODE.
EXEC CICS LINK
  PROGRAM  ("COBSRVI")
  RESP     (CICS-RESP1)
  RESP2    (CICS-RESP2)
  COMMAREA (ERX-COMMUNICATION-AREA)
  LENGTH   (LENGTH OF ERX-COMMUNICATION-AREA)
END-EXEC.
```

> **Note:** After successful call the UOW status is available in the RPC communication area field `COMM-RELIABLE-STATUS`. See *The RPC Communication Area (Reference)* in the COBOL Wrapper documentation.

Step 7b: Examine the Error Code

Check whether the check status call was successful or not.

Step 8: Send a Second RPC Message

Send a second reliable RPC message. See **Step 6a** and **Step 6b**.

Step 9: Check the Reliable RPC Message Status

Check the reliable RPC message before the commit call. See **Step 7a** and **Step 7b**.

Step 10a: Commit both Reliable RPC Messages

Now both reliable RPC messages are committed. This will deliver all reliable RPC messages to the server if it is available.

```
MOVE DFHRESP(NORMAL) TO CICS-RESP1.
MOVE DFHRESP(NORMAL) TO CICS-RESP2.
MOVE "RC" TO COMM-FUNCTION.
MOVE ZEROES TO COMM-RETURN-CODE.
EXEC CICS LINK
  PROGRAM  ("COBSRVI")
  RESP     (CICS-RESP1)
  RESP2    (CICS-RESP2)
  COMMAREA (ERX-COMMUNICATION-AREA)
  LENGTH   (LENGTH OF ERX-COMMUNICATION-AREA)
END-EXEC.
```

Step 10b: Examine the Error Code

Check whether the commit call was successful or not.

Step 11: Send a Third RPC Message

Send a third reliable RPC message. See **Step 5a** and **Step 5b**.

Step 12: Check the Reliable RPC Message Status

Check the reliable RPC message before the rollback call. See **Step 6**.

Step 13a: Roll Back the Third RPC Message

Roll back the current reliable RPC message.

```
MOVE DFHRESP(NORMAL) TO CICS-RESP1.
MOVE DFHRESP(NORMAL) TO CICS-RESP2.
MOVE "RR" TO COMM-FUNCTION.
MOVE ZEROES TO COMM-RETURN-CODE.
EXEC CICS LINK
  PROGRAM  ("COBSRVI")
  RESP     (CICS-RESP1)
  RESP2    (CICS-RESP2)
  COMMAREA (ERX-COMMUNICATION-AREA)
  LENGTH   (LENGTH OF ERX-COMMUNICATION-AREA)
END-EXEC.
```

Step 13b: Examine the Error Code

When the rollback call is returned, check whether it was successful or not. If the rollback call failed, an explicit EOC needs to be sent:

```
MOVE DFHRESP(NORMAL) TO CICS-RESP1.
MOVE DFHRESP(NORMAL) TO CICS-RESP2.
MOVE "RS" TO COMM-FUNCTION.
MOVE ZEROES TO COMM-RETURN-CODE.
EXEC CICS LINK
  PROGRAM  ("COBSRVI")
  RESP     (CICS-RESP1)
  RESP2    (CICS-RESP2)
  COMMAREA (ERX-COMMUNICATION-AREA)
  LENGTH   (LENGTH OF ERX-COMMUNICATION-AREA)
END-EXEC.
```

Step 14a: Perform a Broker Logoff

```
MOVE "LF" TO COMM-FUNCTION.
EXEC CICS LINK
  PROGRAM  ("COBSRVI")
  COMMAREA (ERX-COMMUNICATION-AREA)
  LENGTH   (LENGTH OF ERX-COMMUNICATION-AREA)
  RESP     (CICS-RESP1)
  RESP2    (CICS-RESP2)
END-EXEC.
```

Step 14b: Examine the Error Code

Check whether the logoff call was successful or not.

Writing a Server

There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it returns an error code greater than zero. This causes the transaction (unit of work inside the Broker) to be cancelled, and the error code is written to the user status field of the unit of work. For writing reliable RPC servers, see *Using the COBOL Wrapper for the Server Side*.

To execute a reliable RPC service with an RPC server:

- the parameter logon must be set to "YES", see

    - *Configuring the RPC Server* under *Administering the Batch RPC Server*

    - *Configuring the RPC Server* under *Administering the EntireX RPC Server under z/OS IMS*

    - *Configuring the RPC Server* under *Administering the Micro Focus RPC Server*

    - *Configuring the BS2000/OSD Batch RPC Server* under *Administering the BS2000/OSD Batch RPC Server*

- the parameter LOGN must be set to "YES", see *Configuring the RPC Server*.

4   Reliable RPC for DCOM Wrapper

Writing a Client

All methods for reliable RPC are available on the interface object. See *Standard Wrapper Properties* under *Generated DCOM Wrapper Objects* for details. The methods are

- RPCService.Reliable (put and get)
- RPCService.ReliableCommit
- RPCService.ReliableRollback
- RPCService.MessageID
- RPCService.get_StatusOfMessage

Create Broker object and interface object:

```
// DCOM Wrapper Object
MAILClass    mail;
mail.Logon();
```

Disable reliable RPC:

```
mail.Reliable = mail.RELIABLE_OFF;
```

Enable reliable RPC with AUTO_COMMIT or CLIENT_COMMIT:

```
mail.Reliable = mail.RELIABLE_AUTO_COMMIT;
mail.Reliable = mail.RELIABLE_CLIENT_COMMIT;
```

The first RPC message:

```
mail.SENDMAIL("mail receiver", "Subject 1", "Text 1");
```

Check the status: get the message ID first and use it to retrieve the status:

```
String messageID = mail.MessageID;
String messageStatus = mail.get_StatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The second RPC message:

```
mail.SENDMAIL("mail receiver", "Subject 2", "Text 2");
```

Commit the two messages:

```
mail.ReliableCommit();
```

Check the status again for the same message ID:

```
messageStatus = mail.get_StatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The third RPC message.

```
mail.SENDMAIL("mail receiver", "Subject 3", "Text 3");
```

Check the status: get the new message ID and use it to retrieve the status:

```
messageID = mail.MessageID();
messageStatus = mail.get_StatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

Roll back the third message and check status:

```
mail.ReliableRollback();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
mail.logoff();
```

Limitations

- All program calls that are called in the same transaction (CLIENT_COMMIT) must be in the same IDL library.

- It is not allowed to switch from CLIENT_COMMIT to AUTO_COMMIT in a transaction.

- Messages (IDL programs) must have only IN parameters.

Writing a Server

The server implementation consist of the four classes

- Abstract<IDL library name>Server

- <IDL library name>

- <IDL library name>Server

- <IDL library name>Stub

Add your implementation to the class <IDL library name>Server. There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it throws an exception. This causes a cancel of the transaction (unit of work inside the broker) and the error code is written to the user status field of the unit of work.

5   Reliable RPC for .NET Wrapper

Writing a Client

All methods for reliable RPC are available on the service class object. See description of class `Service` under *Writing Applications with the .NET Wrapper* for details. The methods are:

- Service.SetReliableState
- Service.getReliableState
- Service.ReliableCommit
- Service.ReliableRollback
- Service.GetReliableId
- Service.GetReliableStatus

Example (this example is included as source in folder *examples\ReliableRPC\NetClient*)

Create Broker object and interface object.

```
Mail mail = new Mail();
mail.service.broker.logon();
```

Enable reliable RPC with CLIENT_COMMIT:

```
mail.SetReliableState(Service.ReliableState.RELIABLE_AUTO_COMMIT);
```

The first RPC message.

```
mail.Sendmail("mail receiver", "subject 1", "Text 1");
```

Check the status: get the message ID first and use it to retrieve the status.

```
StringBuilder reliableID = new StringBuilder();
StringBuilder reliableStatus = new StringBuilder();

mail.service.GetReliableID(ref reliableID);
mail.service.GetReliableStatus(reliableID, ref reliableStatus);
Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());
```

The second RPC message.

```
mail.Sendmail("mail receiver", "subject 2", "Text 2");
```

Commit the two messages.

```
mail.service.ReliableCommit();
```

Check the status again for the same message ID.

```
mail.service.GetReliableStatus(reliableID, ref reliableStatus);
Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());
```

The third RPC message.

```
mail.Sendmail("mail receiver", "subject 3", "Text 3");
```

Check the status: get the new message ID and use it to retrieve the status.

```
mail.service.GetReliableID(ref reliableID);
mail.service.GetReliableStatus(reliableID, ref reliableStatus);
Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());
```

Roll back the third message and check status.

```
mail.service.ReliableRollback();
mail.service.GetReliableStatus(reliableID, ref reliableStatus);

Console.Out.WriteLine("Reliable ID = " + reliableID.ToString());
Console.Out.WriteLine("Reliable Status = " + reliableStatus.ToString());

mail.service.broker.logoff();
```

Limitations

1. All program calls that are called in the same transaction (CLIENT_COMMIT) must be in the same IDL library.

2. It is not allowed to switch from CLIENT_COMMIT to AUTO_COMMIT in a transaction.

3. Messages (IDL programs) must have IN parameters only.

Writing a Server

There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it throws an exception. This causes the transaction (unit of work inside the broker) to be cancelled, and the error code is written to the user status field of the unit of work.

6   Reliable RPC for Java Wrapper

Writing a Client

All methods for reliable RPC are available on the interface object. See `RPCService` in the Javadoc documentation of the Java ACI for details. The methods are:

- `RPCService.setReliable`
- `RPCService.getReliable`
- `RPCService.reliableCommit`
- `RPCService.reliableRollback`
- `RPCService.getMessageId`
- `RPCService.getStatusOfMessage`

Example (this example is included as source in the *examples/RPC/reliable/JavaClient* folder):

Create Broker object and interface object.

```
Broker broker = new Broker(Mail.DEFAULT_BROKERID, userID);
Mail mail = new Mail(broker);
broker.logon();
```

Enable reliable RPC with `CLIENT_COMMIT`

```
  mail.setReliable(RPCService.RELIABLE_CLIENT_COMMIT);
```

The first RPC message.

```
  mail.sendmail("mail receiver", "Subject 1", "Text 1");
```

Check the status: get the message ID first and use it to retrieve the status.

```
String messageID = mail.getMessageID();
String messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The second RPC message.

```
mail.sendmail("mail receiver", "Subject 2", "Text 2");
```

Commit the two messages.

```
mail.reliableCommit();
```

Check the status again for the same message ID.

```
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The third RPC message.

```
mail.sendmail("mail receiver", "Subject 3", "Text 3");
```

Check the status: get the new message ID and use it to retrieve the status.

```
messageID = mail.getMessageID();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

Roll back the third message and check status.

```
mail.reliableRollback();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
broker.logoff();
```

Limitations

1. All program calls that are called in the same transaction (`CLIENT_COMMIT`) must be in the same IDL library.

2. It is not allowed to switch from `CLIENT_COMMIT` to `AUTO_COMMIT` in a transaction.

3. Messages (IDL programs) have `IN` parameters only.

Writing a Server

The server implementation consist of the four classes:

- `Abstract<IDL library name>Server`

- `<IDL library name>`

- `<IDL library name>Server`

- `<IDL library name>Stub`

Add your implementation to the class `<IDL library name>Server`. There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it

throws an exception. This causes a cancel of the transaction (unit of work inside the Broker) and the error code is written to the user status field of the unit of work.

7 Reliable RPC for XML/SOAP Wrapper

Writing a Client

The client has to set the parameter `exx-reliable` in the HTTP header or in the XML/SOAP payload. For more information see *Writing Advanced Applications with the XML/SOAP Wrapper*.

Writing a Server

Not applicable.