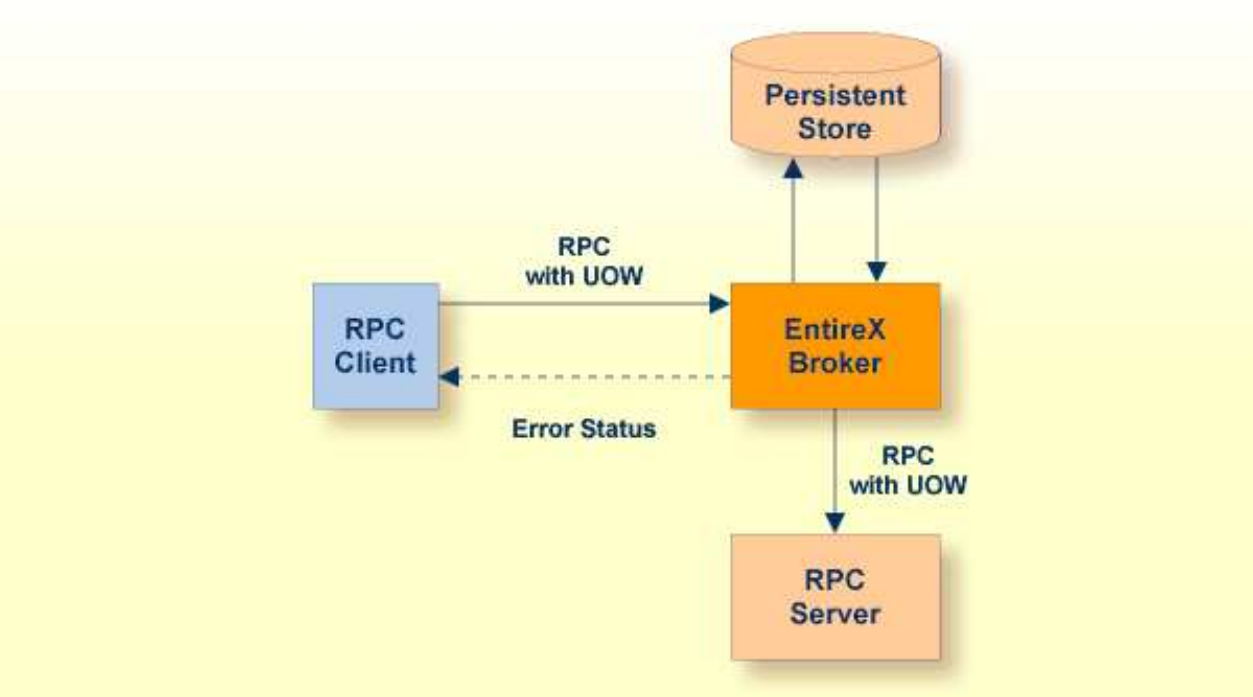
Reliable RPC for C Wrapper

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 EntireX C Wrapper.

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.

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*.

```
/* 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*.

```
/* 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*.

See *Using Persistence and Units of Work*, *Understanding UOW Status* and *Broker UOW Status Transition* in the Platform-independent 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* 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, class-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 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 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 RPC Server* in the EntireX BS2000/OSD administration documentation or *Configuring the RPC Server* under UNIX | Windows

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*.

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`).