Writing Callable RPC Servers with the C Wrapper

The callable RPC server interface enables you to write your own RPC Server. This offers the possibility to integrate RPC servers into third-party systems, as well as to call target servers in programming languages other than C by wrapping them. The programming language for writing a callable RPC Server is C.

This chapter covers the following topics:

Introduction to Callable RPC Servers

The callable RPC server consists of a function containing a loop for recurring RPC execution using a callback technique for user interaction. The function performs all of the necessary communication with the broker such as logon and logoff, service registration, receive and send. The behavior of the callable RPC server can be configured with a server configuration file.

Writing a Callable RPC Server

The main part of the callable RPC server is the C Wrapper runtime function `ERXServingCallback`. This function manages Broker communication as well as the marshalling and unmarshalling of RPC requests using callbacks. The function consists of a loop for RPC execution using callback technique for user interaction. The behavior of `ERXServingCallback` is driven by a configuration file where you set the necessary broker parameters etc. See *Configuring the RPC Server* under UNIX | Windows. In the example below, the name of the configuration file is passed as a parameter to the callable RPC server and given to the `ERXServingCallback` function.

The callback function `ERX_Callback_SERVER_CALL` is called when an RPC request has to be executed. Implement a call to your target server within this callback event. See *Writing the Callback*. The interface of the unmarshalled data given to the callback is compatible with the generated server interface objects for the programming language C. See *Software AG IDL to C Mapping*. Upon return from the callback function, the same applies to parameters replied to the client.

`ERXServingCallback` requires registration of the events before getting control of them. Use `ERXRegisterEvent` with the Event ID `ERX_EVENT_SERVER_CALL` to register the callback function `ERX_Callback_SERVER_CALL` for this purpose.

The criteria to break/stop the RPC execution loop and give control back to the caller can be configured with the configuration file parameter `endworkers`. See *Configuring the RPC Server* under UNIX | Windows. The example below implements a single RPC worker thread within the main function which is ended by a shutdown from outside. Hence use `endworkers=never` as the setting for the configuration file parameter `endworkers` for the example below. This ensures the RPC execution loop is not stopped by broker timeouts or after an RPC request is executed, etc. Use one of the usual ways to stop RPC servers on your platform to stop the callable RPC server. See *Break/Stop the RPC Execution Loop* for more information.

As a general rule before using the RPC C runtime at all, every worker thread must be registered with it using the `ERXRegister` function. `ERXRegister` is therefore the first call to the RPC C runtime and `ERXUnregister` the last. See *Using the RPC Runtime*.

Example

```
void main( int argc, char *argv[ ])
{
   int            bRuntimeRegistered = 0;
   char           myConfigurationFile[512] = "..\\server\\server.cfg";
   void *         myParms = NULL;
   ERXeReturnCode rc = ERX_S_SUCCESS;
   ERX_ERROR_INFO ErrorInfo;
   memset(&ErrorInfo,'\0',sizeof(ErrorInfo));

   /* Treat the input Parameter */
   if( argc == 2)
   {
      strncpy( myConfigurationFile, argv[ 1 ], 512 );
   }
   printf("\nEntireX callable RPC server is running:\n"
            "   Configuration File: %s\n",
              myConfigurationFile);

   /* Register to EntireX RPC Runtime */
   rc = ERXRegister( ERX_CURRENT_VERSION );
   if ( ERX_FAILED(rc ))
   {
      PrintReturnCode(rc,&ErrorInfo);
      goto done; /* ===> */
   }
   bRuntimeRegistered = 1;

   /* Register the Callback Event */
   rc = ERXRegisterEvent(ERX_EVENT_SERVER_CALL,
                         myERX_Callback_SERVER_CALL);
   if ( ERX_FAILED(rc))
   {
       PrintReturnCode(rc,&ErrorInfo);
       goto done; /* ===> */
   }

   /* Execute the Callable RPC Server */
   rc = ERXServingCallback( myConfigurationFile,
                            myParms,
                           (ERX_CF_NOTHING)
   );
   PrintReturnCode(rc,&ErrorInfo);

done:
   if (bRuntimeRegistered == 1)
   {
       ERXUnregister();
   }
   return;
}
```

Refer to *Delivered Examples for the C Wrapper* to locate the example within your installation.

Writing the Callback

This very simple example of a callback implementation uses the `szLibraryName` and `szProgramName` from the `ERX_CALL_INFORMATION_BLOCK` to select requests for the library named `EXAMPLE` and the programs named `CALC` and `SQUARE` in a hard-wired fashion. Other RPC requests are rejected with appropriate error messages. The library and program names correspond to the names given in the IDL file of the calling client.

The focus here is not to show how functions can be called dynamically. Dynamic calling depends on the possibilities of your implementation platform and support by the programming languages in use.

The interface of the unmarshalled data given to the callback is compatible with the generated server interface objects for the programming language C. See *Software AG IDL to C Mapping*. Upon return from the callback function, the same applies to parameters expected by and replied to the client.

- For the IDL data type A, null terminated strings are supplied and expected (corresponding to `DATA_CONV_A=1`).
- For the IDL data types N and P, double is supplied and expected (corresponding to `DATA_CONV_NP=1`).

Returning Errors

With the structure `ERX_ERROR_INFORMATION` either success or failure must always be returned.

User-specific Data

The callable RPC server supports a concept of user-specific data. With this feature it is possible to pass a pointer through the `ERXServingCallback` function directly into the callback. The pointer `myParms`, second parameter of the `ERXServingCallback` in the example above, is available "as is" in the callback here in the first parameter as pointer `pUserInfo`. It can be used, for example, to provide a pointer to a memory location with user-specific data.

Example

```
void myERX_Callback_SERVER_CALL
(
   void                          * pUserInfo,
   ERX_CLIENT_IDENTIFICATION     * pClientInformation,
   ERX_CALL_INFORMATION_BLOCK    * pCallInformation,
   void                          * pParameterArea,
   ERX_ERROR_INFO                * pReturnInfo
)
{
   ERXeReturnCode                rc = ERX_S_SUCCESS;
   printf("\nThis is Callback_SERVER_CALL, serving for %s,%s \n",
          pCallInformation->Callee.szLibraryName,
          pCallInformation->Callee.szProgramName);

   if (strcmp(pCallInformation->Callee.szLibraryName,"EXAMPLE") == 0)
   {
      if (strcmp(pCallInformation->Callee.szProgramName,"CALC") == 0)
      {
         S_CALC  *pParm = (S_CALC *) pParameterArea;
         /* Execute Function */
         pParm->function_result = CALC( pParm->operation,
                                        pParm->operand_1,
                                        pParm->operand_2 );
      }
      else if (strcmp(pCallInformation->Callee.szProgramName,"SQUARE")== 0)
      {
         S_SQUARE  *pParm = (S_SQUARE *) pParameterArea;

         /* Execute Function */
         SQUARE( pParm->operand, &(pParm->result) );

      }
      else
      {
         rc = ERX_E_RPC_CALLEE_NOT_FOUND;
      }
   }
   else
   {
     rc = ERX_E_RPC_LIBRARY_NOT_FOUND;
   }
   pReturnInfo->rc = rc;
   return;
}
```

See *Delivered Examples for the C Wrapper* to locate the example within your installation.

Break/Stop the RPC Execution Loop

The RPC execution loop should normally run continuously until the RPC server is shut down from outside. With the setting of the configuration file parameter `endworkers` you can configure when the RPC execution loop is stopped and control is given back to the caller. See *Configuring the RPC Server* under UNIX | Windows.

The following table explains the `endworkers` parameter.

Value	Explanation
N	**Never** The callable RPC server’s function `ERXServingCallback` breaks/stops the RPC execution loop only if a normal shut down of the RPC server takes place. This setting makes sense with a simple callable RPC server (see *Writing a Callable RPC Server*).
T	**Timeout** The callable RPC server’s function `ERXServingCallback` breaks/stops the RPC execution loop if ● a normal shutdown of the RPC server takes place. ● the time specified by the timeout server parameter has elapsed and no further new RPC request or RPC conversation was active. See *Configuring the RPC Server* under UNIX | Windows. This setting makes sense when working with a scalable number of worker threads. See *Scalable Number of Worker Threads*.
I	**Immediately** The callable RPC server’s function `ERXServingCallback` breaks/stops the RPC execution loop if ● a normal shut down of the RPC server takes place. ● immediately after execution of an RPC request or complete RPC conversation. With this setting you receive control in your callable RPC server after every RPC request or RPC conversation. See *Writing a Callable RPC Server*. You can, for example, use this setting for logging purposes and put a repeat loop around the `ERXServingCallback` function.

Scalable Number of Worker Threads

This section provides some hints on how to implement a callable RPC server with a scalable number of worker threads. This is a more complex server with the ability to clone worker threads to satisfy a high load of client requests.

- Implement a main function registering as an attach server by the broker using `REGISTER, OPTION=ATTACH`. When this server receives attach service requests for clients waiting to be served, start a suitable number of worker threads. See *Implementing an Attach Server*.
- Implement a callable RPC server and its callback to be attached in a thread as described under *Writing a Callable RPC Server* and *Writing the Callback*.
- Use `endworkers=timeout` for the configuration file parameter `endworkers`, if you wish to
  - implement a server that does not exit after the first conversation
  - reduce the number of servers when they are no longer needed

Use `endworkers=immed` if you wish to

- implement a server that handles only one client for one conversation

See *Implementing Servers started by an Attach Server* for more details.