# 27 RPC: The Remote Procedure Calls Library

The HLA Standard Library Remote Procedure Call (RPC) module provides a very easy-to-use mechanism for creating and calling remote procedures. A remote procedure is one that exists in a different process space (possibly even on a different machine). One program (the *local* process) invokes a procedure within a second program (the *remote* process). To a very large degree, the HLA stdlib RPC module lets you define and call remote procedures in a manner quite similar to defining and calling local procedures.

The HLA Stdlib *rpc.hhf* module includes a small compiler for a *mini-language* that processes remote procedure declarations. Once you declare a remote procedure prototype in this mini-language (and the declarations will prove familiar to HLA programmers), very little additional syntax is needed to support remote procedure calls.

## 27.1 Types of Remote Procedures

The HLA stdlib RPC module supports three types of remote procedures: *synchronous*, *bisynchronous*, and *asynchronous*. These three procedure types allow you to balance flexibility and ease-of-use when writing and calling remote procedures.

A synchronous remote procedure comes the closest to resembling local procedure call semantics. When you call a synchronous remote procedure from some local process, that call will not return until the remote procedures returns from the call (and the remote procedure server sends an indication of this to the local process). Synchronous remote procedures are the easiest to understand because they mimic, to a fair degree, the semantics of a local procedure call.

An asynchronous remote procedure call does not block while waiting for the remote procedure to complete execution. Once the local *proxy function*[1] marshalls[2] all the parameters and transmits them to the remote procedure server, the procedure immediately returns and program execution continues in the local process. Asynchronous procedure calls are generally much faster (on the local machine) because the asynchronous procedure call doesn't have to way for the remote procedure to finish (and send an acknowledgement over the network, which can be slow).

Bisynchronous procedures are a combination of synchronous and asynchronous procedures. The initial call is asynchronous, but there is a special function that you can call that will block the local process until the remote procedure returns (and the remote server acknowledges this). Bisynchronous procedures have one additional benefit not present in the other two remote procedure forms: you can return function results from a remote procedure when using bisynchronous calls.

## 27.2 Remote Procedure Call (RPC) Protocol

The HLA stdlib RPC module supports a peer-to-peer remote procedure call protocol. This protocol is built upon network sockets and is OS independent (that is, the HLA stdlib RPC code does not rely upon any particular RPC facilities provided by a specific operating system). The protocol operates in a dual client/server architecture. There are two processes involved in a remote procedure call; the server process is where the remote procedures reside, the client process calls the remote procedures. The protocol is a dual client/server model because the client process also runs a background server to receive responses and acknowledgements from the server. For this reason, a remote procedure server can only service one client at a time (effectively making this a peer-to-peer architecture).

Note that the RPC server code runs in a separate thread on the server machine. This means that in addition to linking in the socket library, code that uses remote procedures must also link in the threads module (and, therefore, you must supply the "-thread" command-line parameter on all modules you compile).

This architecture forces some important limitations. First of all, as just noticed, a set of remote procedures under the control of a remote server can only be called by one client until the client gives up the network connection to the server. This limitation isn't as bad as it seems because it's perfectly possible to run multiple RPC servers within the same process. So one program could be running multiple RPC servers and serving multiple clients, though this is not expected to be a common usage of the HLA RPC module.

The RPC server serializes all calls made to it. Although the local process can run multiple threads, with each thread making remote procedure calls, the calls are received one at a time in the server and queued up for execution. The server retrieves on request at a time, executes the associated code, acknowledges the call (if

1. A proxy function is a local procedure that is stand-in for the remote procedure. Calling a proxy function uses the same syntax as the actual function; the proxy collects all the arguments (marshalls them) and transmits the arguments to the remote procedure server for execution.
2. Marshall, in this context, means to collect the arguments into a single package.

synchronous or bisynchronous), and the fetches and processes the next call request. One advantage to this approach is that the remote procedures can assume that they are not executing concurrently (with respect to one another). If concurrent execution is an absolute necessity, the solution is the same as for multiple clients -- just run multiple RPC servers in the remote process and make the calls to different RPC servers; those calls will run concurrently (and it will be your responsibility to synchronize access to shared objects).

On the local process side, the HLA RPC module serializes calls to a single remote procedure. That is, if two different threads call the same remote procedure call (at the same time), the RPC library will block one of the two procedure calls until the other returns. For asynchronous RPCs, this is only as long as it takes to marshall the parameters and transmit them to the remote server. For synchronous and bisynchronous calls, the second call is blocked until the first one is acknowledged. In theory, this could also be fixed by running multiple servers (calling the same code on the remote server), but this would create some synchronization problems you'd have to handle manually.

Although a major goal (indeed, the main goal) of the HLA stdlib RPC module is to minimize the semantic differences between local and remote procedure calls, there are some things that are impractical to simulate in a remote procedure call and return. For example, changes to register values on the remote machine will not be reflected in the registers in the local process (in theory, it might be possible to do this, but it would be too expensive to do so). Likewise, because the local and remote processes run in different address spaces (indeed, usually on different machines) passing parameters by reference (or by any other scheme other than by value) is impractical.

# 27.3 The RPC Declaration Language (RDL)

The primary goal of the HLA stdlib RPC module is to minimize the (syntactical) differences between remote and local procedure calls. Largely, this is accomplished via the RPC Declaration Language (RDL). You make the RDL available to your programs by including the *rpc.hhf* header file at an appropriate spot at the beginning of your source file (e.g., near the other #include statements in your program). This makes all the macros, on which the RDL is built, available to your program. For example:

```
program t;
#includeOnce( "stdlib.hhf" )
#includeOnce( "rpc.hhf" )
  .
  .
  .
end t;
```

Note that *rpc.hhf* automatically includes the *sockets.hhf* and *threads.hhf* header files. As noted earlier, this means that you will need to supply the *-thread* command-line parameter when compiling all HLA files in the project. If you attempt to compile a source file that includes *rpc.hhf* without supplying the *-thread* command-line parameter, HLA will emit a warning complaining about its absence.

The RDL uses the following basic syntax (these statements must appear after you've included the *rpc.hhf* header file):

```
remoteProcedures( classNamePrefix )

  << Synchronous, asynchronous, and bisynchronous procedure declarations >>

endRemoteProcedures
```

The *classNamePrefix* argument specifies the client and server names. The `remoteProcedures` statement will generate a couple of classes named *`classNamePrefix`*`_server_t` and *`classNamePrefix`*`_client_t`. It will also create a couple of instance variables named *`classNamePrefix`*`_server` and *`classNamePrefix`*`_client` (note the lack of a "_t" on these instance variable names). These class/objects are singletons; that is, there is only one instance variable associated with each class (it doesn't really make any sense to have multiple objects of these classes). You will sometimes refer to these singleton instance variables (and the typenames) within your programs, but most of the time you will not use them; most of the work they do will be behind your back.

Within the body of the `remoteProcedures..endRemoteProcedures` statement are the actual remote procedure declarations. These take one of the following three forms:

```
sync( <<procedure declaration>> )
async( <<procedure declaration>> )
bisync( <<procedure declaration>> {:optional_return_type} )
```

A *<<procedure declaration>>* looks very similar to an HLA procedure declaration. It takes one of the following two forms:

*procedureID*
*procedureID*( *<<optional parameter list>>* )

*procedureID* is a (unique) HLA identifier that specifies the remote procedure's name. The *<<optional parameter list>>* item is a list of (pass by value) HLA parameter declarations whose syntax is identical to a normal formal parameter list declaration (except only implicit pass by reference is allowed). Here are some examples of legal remote procedure declarations:

```
sync( proc1 )                                   // No parameters
bisync( proc2( i:int32; j:uns32; k:real32 ) )   // Three arguments
async( proc3( x:string ) )                      // One argument
```

There are some severe limitations on the types of arguments you can pass to a remote procedure. In particular, you can only pass arguments of the following types:

- byte, boolean, char, uns8, int8
- word, uns16, int16
- dword, uns32, int32, real32
- qword, uns64, int64, real64
- tbyte, real80
- lword, cset, uns128, int128
- string
- blob_t (note: blob.t, blob.blob, and blob.blob_t will *not* work)

In particular, note that you cannot pass composite types such as arrays or records. We'll see how to manually pass these types of arguments a little later in this document. You cannot pass pointers or thunks to a remote procedure (which wouldn't make any sense because the remote procedure executes in a different address space). Of course, you could cast a pointer or thunk to a dword or qword, but keep in mind that the addresses in the remote procedure won't match those in the local procedure, so doing so will likely result in crashing the system or producing weird results. Remember, all arguments must be passed by value to a remote procedure.

Bisynchronous remote procedures support an optional return result. In the example immediately above, the bisynchronous procedure `proc2` does not have a return result associated with it. The following examples demonstrate some bisync declarations with a return result:

```
bisync( func1( x:real32 ):real64 )
bisync( rmtFileName:string )
bisync( rmtAppend( s1:string; s2:string ):uns64 )
```

The set of return types are also limited to the basic types listed earlier. Again, returning pointers and thunks is a no-no and there is no direct support for returning a composite data type. We'll look at ways to return composite data structures later in this document.

Here is a complete `remoteProcedures..endRemoteProcedures` statement using the declarations given thus far::

```
remoteProcedures( classNamePrefix )

  sync( proc1 )
  bisync( proc2( i:int32; j:uns32; k:real32 ) )
  async( proc3( x:string ) )
  bisync( func1( x:real32 ):real64 )
  bisync( rmtFileName:string )
  bisync( rmtAppend( s1:string; s2:string ):uns64 )
```

```
endRemoteProcedures
```

Generally, you're going to place the remoteProcedures..endRemoteProcedures statement within its own header file. For example, the code above might appear as part of the *myRPCs.hhf* header file:

```
// myRPCs.hhf:

#ifdef( !@defined( rpc_hhf ))
  ?rpc_hhf := true;

  #includeOnce( "rpc.hhf" )   // May as well do this here

  remoteProcedures( classNamePrefix )

      sync( proc1 )
      bisync( proc2( i:int32; j:uns32; k:real32 ) )
      async( proc3( x:string ) )
      bisync( func1( x:real32 ):real64 )
      bisync( rmtFileName:string )
      bisync( rmtAppend( s1:string; s2:string ):uns64 )

  endRemoteProcedures

#endif
```

The reason you'll want this in a header file is because you have to include this code in two files: in your local client program and in the remote server program.

If you stick anything besides a `sync, bisync,` or `async` statement between the `remoteProcedures` and `endRemoteProcedures` clauses, The RDL will ignore those statements and pass them straight through to HLA. Because the RDL statements do not directly emit any code, this is equivalent to placing those statements immediately before or after the `remoteProcedures..endRemoteProcedures` statement. Generally, it's bad style to do this; if you have a good reason for wanting to emit some code (or declarations) at that point, you can just as easily place them before the `remoteProcedures` statement.

As just noted, the RDL statements do not directly emit any code. Instead, they (effectively) create two header files containing the code they produce. These two header files are named *rpc_client_implementation.hhf* and *rpc_server_implementation.hhf*, respectively.[3] You should include the rpc_client_implementation.hhf header file in the local client code (generally immediately after including the header file containing the RDL code) and you should include the rpc_server_implementation.hhf header file in the server source file, e.g.,:

```
program rpcServer;
#include( "stdlib.hhf" )
#includeOnce( "myRDLcode.hhf" )
#includeOnce( "rpc_server_implementation.hhf" )
      .
      .
      .
  << remainder of RPC server program >>

end rpcServer;
```

And for the client:

3. Technically, the RDL statements do not create these header files. The RDL statements create two text strings that these header files expand. You could expand those strings directly rather than including the header files, but there are technical reasons (dealing with error reporting) for expanding them in these header files.

```
program rpcClient;
#include( "stdlib.hhf" )
#includeOnce( "myRDLcode.hhf" )
#includeOnce( "rpc_client_implementation.hhf" )
      .
      .
      .
  << remainder of RPC client program >>

end rpcClient;
```

Note that you cannot include both RPC implementation header files in the same source file. This will result in duplicate symbol definition errors.

# 27.4 RPC Preliminaries

The HLA stdlib RPC module works across TCP/IP using the HLA sockets library. This means that you need two things in order to establish communication between a local client and a remote procedure server: an IP address (of the server) and a pair of unused socket port numbers. It is possible to run the remote procedure server on the same machine as the local client (this is actually a very useful configuration for testing purposes), though you will typically run the remote server on a separate machine. After all, if you're always calling the remote procedures on the same machine as the local client, it would probably be more efficient to use threads rather than remote procedure calls. The HLA examples download contains an example of an RPC client and server that both run on the same machine. They call the sock.hostAdrs function to get the IP address of the machine they are running on. In general, however, it is your responsibility to determine the IP address of the machine running the RPC server.

Note that the server doesn't need to be explicitly given the IP address of the client. When the client connects with the server, it automatically transmits its own IP address to the server, so the server can use that address to connect make the second client/server connection back to the client.

The RPC protocol uses two consecutive socket port numbers. You supply only one port number and the RPC code computes the second value by adding 1 to the value you supply. One port is used for communication between the local client and the RPC server, the other port number is used for communication between the server and the client. Generally, any port number greater than $8000 that doesn't conflict with other software you're currently running is fine. The example in the HLA examples download uses port number $9998 (and $9999), but this value was chosen at random and has no official meaning.

Because the RPC system uses sockets, your client and server programs must initialize the socket subsystem by calling the HLA stdlib procedure `sock.socketInit`. After that, you're ready to create the client or server object and get things running.

# 27.5 RPC Clients

As noted earlier, the `remoteProcedures..endRemoteProcedures` statement will automatically construct two classes (one for the server module and one for the client module). You control the name of the classes via the argument you supply to `remoteProcedures`, e.g.,:

```
remoteProcedures( myRPC )

    sync( proc1 )
    bisync( proc2( i:int32; j:uns32; k:real32 ) )
    async( proc3( x:string ) )
    bisync( func1( x:real32 ):real64 )
    bisync( rmtFileName:string )
    bisync( rmtAppend( s1:string; s2:string ):uns64 )

endRemoteProcedures
```

This declaration, in the client file (that includes *rpc_client_implementation.hhf*), creates a class named `myRPC_client_t`. It also creates the VMT for that class and a single `static` object instance of that class named `myRPC_client`. Like any statically allocated class object, you'll need to initialize that object before you use it. You accomplish this by calling the `myRPC_client.create` constructor for the class. This constructor has two arguments: the IP address of the remote server and the port number (first of two) that the server will be listening on for the client.

The constructor actually does a lot more than simply initialize the class object. It actually connects to the server. It will not return until a connection is made with the server process. Upon return from the constructor, you're ready to make some remote procedure calls. Here is a typical invocation of the constructor:

```
myRPC_client.create( ipAdrs, $9998 );
stdout.put( "Connected to RPC server" nl );
```

When you compile the RDL statements (in the *myRDLcode.hhf* file in the earlier examples), the RDL compiler generates a considerable amount of code and places it in the `rpc_client_implementation.hhf` header file. For all remote procedure types, the RDL compiler will generate a local *proxy function* that has the same name and calling sequence as your declared remote procedures. For the current example, you wind up with procedure prototypes like the following:

```
procedure proc1; external;
procedure proc2( i:int32; j:uns32; k:real32 ); external;
procedure proc3( x:string );
procedure func1( x:real32 ); external;
procedure rmtFileName; external;
procedure rmtAppend( s1:string; s2:string ); external;
```

The RDL compiler actually writes the code for these functions (you compile this code into your client program by including the *rpc_client_implementation.hhf* header file). These proxy functions collect (marshall up) any arguments and send a packet to the remote server identifying the procedure to run and supplying the arguments for that call.

On the client side, once you've initialized the `myRPC_client` object, you're free to call these procedures just as though they were local procedures.

For synchronous and asynchronous remote procedures (`proc1` and `proc3` in this example), that's all there is to it. You call the proxy procedure and the code executes on the remote server. For synchronous procedures (e.g., proc1), the call doesn't return until the code completes execution on the server (and the server notifies the client of the completion). For asynchronous procedure calls (e.g., `proc3`), the call returns immediately after the proxy procedure marshalls the arguments and ships them off to the remote server; asynchronous calls to not wait for the completion of the code on the remote server.

Note that the RDL compiler will create an additional client-side procedure in addition to the proxy functions for synchronous procedures. This procedure will have the same base name as the proxy procedure with "`_return`" appended to the name (e.g., `proc1_return` in the current example). This procedure is for internal use only. You must never call this procedure; doing so may make the system unreliable. Note that the RDL compiler does not generate any extra procedures for asynchronous procedure declarations. On the client, the proxy function is the only code the RDL compiler generates for an asynchronous procedure.

Bisynchronous procedures are considerably more complex than synchronous or asynchronous procedures. Like asynchronous and synchronous procedures, the RDL compiler will emit a proxy function that marshalls all the arguments and ships them over to the remote server for execution. Like asynchronous procedures, this proxy function will immediately return after it transmits the arguments to the remote server; it will not wait for the completion of the remote procedure call. Like synchronous procedures, the RDL compiler will create a "`_return`" procedure (which you must never call) that the server remotely invokes on the client when the remote procedure completes execution.

The RDL compiler generates one additional client-side function for bisynchronous procedures: a "`_waitForReturn`" function (that has the proxy name prepended to it, e.g., `func1_waitForReturn`). The "`_waitForReturn`" procedure serves two purposes: it delays the client program until the remote procedure completes execution and it provides a mechanism for retrieving a bisynchronous function return result.

Unlike the "`_return`" function, a client-side application must call the "`_waitForReturn`" function at some point after calling the proxy function. **This is absolutely required! If you fail to call the corresponding "`_waitForReturn`" function after a bisynchronous procedure call, you will deadlock (hang) the system if you make a second call to that same proxy function**. Calling the proxy function enters a critical section associated with the remote procedure and calling the corresponding "`_waitForReturn`" function leaves that critical section. If you attempt to call the proxy function twice without an intervening "`_waitForReturn`" call, you will attempt to reenter the critical section (in the same thread) and this may produce deadlock. Moral of the story, always be careful when using bisynchronous procedures and ensure that you call the "`_waitForReturn`" procedure as quickly as is reasonable.

Bisynchronous procedures are quite useful for continuing to do some work while you're waiting for the remote procedure to finish execution (and, generally the slower activity, waiting for the client and server to communicate the call and its completion between themselves). You can call a bisynchronous procedure, do some

work (that doesn't depend on the completion of that procedure), and then call the "_waitForReturn" procedure when you need to synchronize the execution of the local client code with the remote server.

Another reason for bisynchronous procedure calls is to return function results from a remote procedure. If you specify a return type when declaring a bisynchronous procedure, e.g.,:

```
bisync( func1( x:real32 ):real64 )
```

Then the RDL compiler will generate a func1_waitForReturn procedure with the following prototype:

```
procedure func1_waitForReturn( var rtn:real64 );
```

Note that this has a single pass-by-reference argument whose type is the same as the bisync return type. Calling such a typed "_waitForReturn" function will store the remote function's return result in the variable you pass as the procedure's parameter.

String and blob_t types are special cases. Consider the rmtFileName function from the earlier examples and its "_waitForReturn" function:

```
bisync( rmtFileName:string )

// Return function prototype:

rmtFileName_waitForReturn( var rtn:string );
```

The client-side remote procedure code automatically allocates storage for a string (or blob) result on the heap and stores a pointer to that new string (or blob) in the variable you pass to rmtfileName_waitForReturn. **This function does not store the string or blob data in the object you pass; it overwrites the variable's data pointer with the pointer to the new string or blob on the heap.**

These semantics have two ramifications in your program. First of all, if the string variable you pass to rmtFileName_waitForReturn already points at a string on the heap and you do not have another copy of that pointer laying around, then you will have a memory leak upon return from rmtFileName_waitForReturn when the original pointer is overwritten by the new one. Second, because the remote procedure call system allocates the storage for the return string on the heap, it is your responsibility to free this storage when you are done using the string (by calling str.free for strings or blob.free for blobs).

When you are done using the remote procedure server, you can shut down the connection by calling the destructor for the client class object. For example, if you're using the name myRPC (as in the earlier examples that described the constructor), you can shut down the system with the following call:

```
myRPC.destroy();
```

Generally, it's a good idea to wait a second or two after calling the destructor to give the remote system time to shut down before you kill the socket connection (this is optional, it's not strictly required). You can do this with an HLA stdlib call such as os.sleep(1);

## 27.6 RPC Servers

The server side of a remote procedure call client/server pair is slightly more complicated than the client side. This is largely because in addition to initializing (and destroying) the server object, you've actually got to supply the remote procedures that the client will be calling. As on the client side, synchronous and asynchronous procedures are fairly easy to understand -- they look and behave much like local procedures; bisynchronous procedures, on the other hand, introduce some additional complexities because of return results and their bisynchronous nature.

Like the client side, the first thing you'll find of interest in a remote procedure server program is the inclusion of the RDL source code and the associated implementation file. Continuing the example from the previous system, here's a reminder of what the basic program file will look like:

```
program rpcServer;
#include( "stdlib.hhf" )
#includeOnce( "myRDLcode.hhf" )
#includeOnce( "rpc_server_implementation.hhf" )
      .
```

```
        .
        .
    << remainder of RPC server program >>

end rpcServer;
```

The structure of the main program is going to be somewhat different from the client. This is because the server exists primarily to provide remote procedure call services to the client. The client, on the other hand, exists to solve some application problems and remote procedure calls might only be a tiny part of the work that takes place on the client. On the server, however, the work revolves totally around providing that remote procedure call service.[4] A typical main program in an RPC server will consist of two calls: one call to the server's class constructor and one call to its destructor.

When you compile the RDL code (i.e., the `remoteProcedures..endRemoteProcedures` statement), the RDL compiler generates a class and an object instance variable specifically for the server application. Assuming you've specified the name `myRPC` in the remoteProcedures statement, the class name will be `myRPC_server_t` and the object instance will be `myRPC_server`. The constructor for this new class has the following calling syntax:

```
myRPC_server.create
(
    basePortNumber,
    &serverConnectedProcedure,
    timeoutThunk
);
```

The `basePortNumber` argument is the port number (first of two). This must match the value you use on the client side. Remember that the RPC server and client use two consecutive port numbers. So if you specify $9998 as the port number, the system will actually use ports $9998 and $9999.

The `serverConnectedProcedure` argument is the name of a parameterless procedure that the RPC server will call when the client successfully connects when the server and the server, in turn, successfully connects with the acknowledgement server on the client side. There is one requirement for this procedure: it must call the "`connected`" method of the "`*_server_t`" class created by the RDL compiler. Assuming the use of the `myRPC` argument to remoteProcedures, here's what the minimal `serverConnectedProcedure` will look like:

```
procedure serverConnectedProcedure;
begin serverConnectedProcedure;

  myRPC_server.connected();

end serverConnectedProcedure;
```

Though this is the minimal amount of work this procedure should do, there are some minor embellishments that are useful. Here is a typical example (taken from the RPC example in the HLA examples download):

```
static
     quitServer :boolean;


procedure serverConnected;
begin serverConnected;

  stdout.put( "Client connected with server" nl );

  // Start the real server code:

  myRPC_server.connected();
```

4. Technically, there is no reason you couldn't start up a thread in the server program and run the remote procedure call server in that thread while the main thread does other work, but generally your system will be more robust if you limit the activities of the remote procedure call server to just handling remote procedure calls.

```
    stdout.put( "Client disconnected from server" nl );
    mov( true, quitServer );

end serverConnected;
```

Printing "Client connected with server" and "Client disconnected from server" are obvious modifications (note that when `myRPC_server`.connected returns, the client and server have disconnected). The reason for setting the global variable `quitServer` to true will become apparent in a few moments.

The last argument to the server class constructor is a "timeout thunk." If you're unfamiliar with thunks, just note that they are a fancy form of a procedure that you can declare in-line in some other code. The cool thing about thunk parameters (as opposed to procedures you pass by address as a parameter) is that you can encoded the thunk statements in-line in the argument list of a procedure call.

The constructor calls the `timeoutThunk` on a periodic basis (the interval is set by the thunk itself). On entry into the server's constructor thunk, EAX points at an hla.timeval variable that the thunk can use to change the timeout period. On return, EAX contains false if the constructor is to continue waiting for the server to connect, EAX should contain true if a timeout has occurred and you want to return from the constructor without connecting. Here's a typical constructor call with an in-line thunk:

```
    mov( false, quitServer );
    myRPC_server.create
    (
        $9998,
        &serverConnected,
        thunk
        #{
            // On entry to thunk, EAX contains the address of the timeout
            // variable. Set this as desired for the timeout (0.1 second,
            // in this case).

            if( eax <> NULL ) then

                // Timeout is 0.1 seconds while waiting for
                // connection:

                    mov( 100_000, (type hla.timeval [eax]).tv_usec );

            endif;
            movzx( quitServer, eax );

        }#
    );
```

Notice that this thunk returns the value of `quitServer` in EAX. Because this code sequence also initializes `quitServer` with false (and no other code in the thunk ever sets it to true), this code will never time out. The `myRPC` constructor will continually call this thunk at 0.1 second (100,000 microsecond) intervals until it connects with a client.

When the client calls its destructor, the client sends a message to the server telling it to disconnect. Whenever a client disconnects from the server, the server reenters the loop waiting for another client to connect with it. However, if you look back at the `serverConnected` code given earlier, it sets the `quitServer` global variable to true, so the first time the constructor reenters the thunk, the thunk returns true in EAX and the server quits execution.

Upon returning from the constructor, it's a good idea for the server to delay a couple of seconds before calling the class destructor method (`myRPC_server.destroy`) and quitting the program:

```
    // Short delay to allow all transmissions to complete before we bail:

    os.sleep( 2 );

    my_server.destroy();
```

All that remains in the server module is to write the actual remote procedures that the client will call. As for the client side, asynchronous and sychronous procedures are relatively straight-forward and look just like local procedure calls. For example:

```
procedure proc1;
begin proc1;

  stdout.put( "Client called synchronous procedure proc1" nl );

end proc1;

procedure proc3( x:string );
begin proc3;

  stdout.put( "Client called asynchronous procedure proc3, x=", x, nl );
  str.free( x );

end proc3;
```

This is relatively straight-forward stuff. About this only issue of which you must be aware (and `proc3` demonstrates) is that when you pass string values as parameters, the RPC code allocates storage for the string object on the heap and it is your responsibility to free that storage when you are done using the string's data.

Behind your back, the RDL compiler generates some additional functions for all procedures (synchronous, asynchronous, and bisynchronous). These procedures have the same name as the primary procedure with the addition of a "`_marshall`" suffix. The marshall procedures read the arguments (if any) from the network, save them into local variables, and then call the actual user-written procedure with these arguments. It should go without saying that user-written code should never call these marshalling procedures; the marshalling procedures expect data coming across the network and they must only be called from the RPC server in response to an RPC request from the client.

As on the client side, bisynchronous procedures are slightly more complex than asynchronous and synchronous procedures. Let's first consider the `proc2` example from earlier in this document:

```
bisync( proc2( i:int32; j:uns32; k:real32 ) )
```

Here is a simple implementation of `proc2`:

```
procedure proc2( i:int32; j:uns32; k:real32 );
begin proc2;

  stdout.put( "proc2 was called, i=", i, ", j=", j, ", k=", k, nl );
  proc2_return();

end proc2;
```

The important thing to notice in this code is the call to the `proc2_return` procedure. This is an RPC compiler generated procedure that notifies the client when the procedure is done executing. Generally, you must call this procedure at the end of a bisynchronous procedure (that's the logical place to call this procedure, though there is nothing stopping you from calling it earlier if you have a good reason to do so). **You must call this function exactly once in every bisynchronous procedure**. If you don't call it, the client will hang up waiting for the bisynchronous procedure call to finish when it calls the corresponding `proc2_waitForReturn` function.

Now consider an example of a bisynchronous remote procedure that has a function return result:

```
//   bisync( rmtFileName:string )

procedure rmtFileName;
begin rmtFileName;

  // Code to compute the filename and produce the string
  // value "fileName"
```

```
    rmtFileName_return( fileName );

end rmtFileName;
```

Whenever a bisynchronous procedure returns a function result, the corresponding "*_return" function will require a single (pass by value) parameter whose type matches the return type. Note that the return funciton will completely transmit the data before it returns. Therefore, you are free to destroy the object (e.g., free the storage associated with the `fileName` string variable) upon return from the return function.

# 27.7 Passing Large Objects Between the Client and Server

The existing RDL does not support passing composite data types (other than strings and blobs) to and from remote procedures. In this section you'll see how to overcome this limitation. You can pass large data types between a client and server, however, you'll have to manually pass that data yourself rather than relying on RDL compiler generated code to do the job for you.

The RPC2 project in the HLA examples download (which will be reproduced here) demonstrates how to pass large chunks of data between a client and an RPC server. Passing a large data structure to the RPC server is the easiest to understand, so we'll start with that explanation.

There are two ways to pass large chunks of data: using explicit networking calls or moving the data to a blob and transmitting the blob. We'll start with a discussion of using explicit networking calls.

Although the RPC protocol passes fixed amounts of data between the client and the server, this protocol is built on top of the HLA sockets library, that allows you to transmit an arbitrary amount of data from one network node to another. Inside the classes that the RDL constructs are two HLA socket objects: a `client_t` object and a `server_t` object, named `client` and `server`, respectively. As long as you are careful, you can use these `client` and `server` objects to communicate data from the client application to the RPC server application.

Before discussing how to pass a block of data from the client to the server using explicit networking calls, an important warning is worth mentioning: the RPC server process has a very strict data transmission and receipt protocol. You will be injecting bytes into the (client) transmission stream and intercepting bytes in the (client) reception stream. The code you write on the server and client sides to handing these extra bytes must transmit and receive the exact number of bytes your code on the other side of the network is expecting. The HLA stdlib RPC module is not fault tolerant with respect to recovering from a protocol mish-mash (TCP/IP guarantees correct delivery, it doesn't make sense to replicate the error checking in the RPC code). If you transmit M bytes to the server but your code on the server only reads (M-N) of those bytes (or attempts to read M+N bytes), then the RPC protocol will be out of sync and unexpected results (usually a crash) will happen shortly thereafter. Moral of the story: if you transmit M extra bytes from one network node to the other, make sure the other reads those M bytes (and no more).

If you want to transmit a large block of bytes from the client to the RPC server using explicit networking calls, the first thing you're going to need to do is to create a bisynchronous procedure to handle the transmission. This *has* to be a bisynchronous procedure. The reason for a bisynchronous procedure is because the code you write on the server side will need to read the extra bytes you send it before the RPC server attempts to read another command from the network socket. Bisynchronous procedures give your application complete control over this process on the server as well as complete control over the acknowledgement receipt on the client side. Synchronous procedures won't work because the client-side call will wait for an acknowledgement before you get control back (so you won't be able to send the array data until after the server has acknowledged the call and has already started waiting for a new command). Asynchronous procedures could work, but it's easier to mess up the RPC protocol with asynchronous procedures, so bisynchronous procedures make a better choice.

The first place to start is with the data structure you want to pass from the client to the server. Generally, it's a very good idea to place a type definition for this data type in the same header file that contains your RDL code (that is, the `remoteProcedures..endRemoteProcedures` statement). For this section's example, we're going to pass a dword array with 16 elements (that is, a 64-byte object). Here's the data type declaration that appears in the *sc.hhf* header file in the RPC2 example:

```
type
     array_t   :uns32[16];
```

By convention (a convention I'm creating as I write this, as this is the first example of this process ever written), the remote procedure on the RPC server will expect a single dword parameter that will hold the size of the object. Technically, no such parameter is necessary because the data type (`array_t` in this example) is visible on both the client and server sides and both sides can easily compute the size of the object using the `@size( array_t )` compile-time language function invocation. However, by explicitly passing the size, you

can run a consistency check on the server side just to verify that there aren't any problems. Here's what a typical RDL statement for a procedure with a large array argument might look like:

```
bisync( passBig( size:dword )   )
```

On the client side, you're going to need to write a separate function (separate from the one the RDL compiler generates for `passBig`) that you can use to pass the array to the server. This new procedure will call `passBig` to get the server side process running and then it will transmit the array data to the server before waiting for the server to return an acknowledgement. In the RPC2 example, I've called this procedure `passArray`. Here is the code for `passArray`:

```
procedure passArray( var a:array_t );
var
    b:blob_t;
begin passArray;

    push( esi );
    push( edi );

    passBig( @size( array_t ));
    lea( esi, myRPC_client.client );
    (type client_t [esi]).write( val a, @size( array_t ));
    passBig_waitForReturn();

    pop( edi );
    pop( esi );

end passArray;
```

The call to `passBig` in this procedure kicks off the data transmission process. The `passBig` procedure on the server side will accept the size argument and then the user-written code in that procedure will wait for the arrival of the data. When the server receives all the data, it will send an acknowledgement back and the `passwBig_waitForReturn` function will return and the client will continue execution.

Upon entry into the `passBig` procedure on the server, we've already received the `size` parameter value and the client has probably sent the bulk data on its way as well. Therefore, one of the first things we've got to do is to read that block of data from the network socket. Once we've read all the network data, we have to call the `passBig_return` procedure to send an acknowledgement to the client that the procedure is done executing. Here's the complete code for the `passBig` procedure:

```
procedure passBig( size:uns32 );
var
    anArray :array_t;

begin passBig;

    assert( size = @size( array_t ));
    lea( esi, myRPC_server.server );
    (type server_t [esi]).read( anArray, @size( array_t ));
    stdout.put( "Received anArray:" nl );
    for( mov( 0, ecx ); ecx < @elements( array_t ); inc( ecx )) do

        stdout.put( "anArray[", (type uns32 ecx), "]=", anArray[ecx*4], nl );

    endfor;
    passBig_return();

end passBig;
```

Notice how the call to passBig_return occurs after we've read all the data from the client.

The second way, and arguably the standard way, to pass a large data type from the client to the RPC server is by passing a `blob_t` argument. On the client side, you would create a `blob_t` object, store the large data object into the blob, and then remotely call the desired procedure passing the blob argument.

Before demonstrating how this is done, you should be aware of an important fact: if you intend to pass a blob object between the client and the server you must use the `blob_t` data type because the RDL compiler recognizes only this blob type. Although `blob.t, blob.blob_t,` and `blob.blob` are usually synonyms for blob_t, the RDL compiler doesn' recognize these synonyms (for syntactical reasons).

One advantage of passing large data types as blobs is that you can create syncrhonous, bisynchronous, and asynchronous remote procedures without any trouble. In this example, we'll use a synchronous procedure declaration. From the *sc.hhf* header file (in the RPC2 project in the HLA examples downloads) you'll find the following RDL declaration for the `passBigb` procedure:

```
sync( passBigb( b:blob_t ) )
```

In the client source file, you'll need to write a procedure that will collect your large object's data together and place that information into a blob (assuming the large object isn't a blob to begin with). Then you will call the remote procedure and pass that blob as an argument. When you're done with the blob you've created, you will typically free its storage before returning. Here's the implementation of `passArray2` from the RPC2 project:

```
procedure passArray2( var a:array_t );
var
    b:blob_t;
begin passArray2;

    blob.alloc( @size( array_t ));
    mov( eax, b );
    blob.write( b, val a, @size( array_t ));
    passBigb( b );
    blob.free( b );

end passArray2;
```

First, this code allocates a blob object large enough to hold the data object to pass to the remote procedure. Then it writes the data (from the array in this example) to the blob. Then it calls `passBigb` to pass the array data, encapsulated in the blob, over to the server. Finally, it deletes the storage associated with the blob.

On the server side, the `passBigb` procedure receives the blob data via a blob -typed parameter. This procedure simply has to read the data out of the blob and place it in the local array data object. Whenever you pass a blob object to a remote procedure, that remote procedure is required to free the storage associated with that blob (same as the situation for strings). Here's the server side code from the RPC2 example that demonstrates this activity:

```
procedure passBigb( b:blob_t );
var
   bArray  :array_t;

begin passBigb;

   blob.length( b );
   assert( eax = @size(array_t) );

   lea( eax, bArray);
   blob.read( b, [eax], @size( array_t ));
   stdout.put( nl "Received bArray:" nl );
   for( mov( 0, ecx ); ecx < @elements( array_t ); inc( ecx )) do

      stdout.put( "bArray[", (type uns32 ecx), "]=", bArray[ecx*4], nl );

   endfor;
```

```
    blob.free( b );

end passBigb;
```

To return a large data object as a function result there is only one mechanism available: pack the data into a `blob_t` object and return that blob to the client (and unpack the blob when the client receives it). As is the case for all remote procedure that return a value, you must use a bisynchronous procedure to achieve this. Here's the RDL declaration from the RPC2 example program for a `rtnBig` procedure that returns a 16-element array object:

```
bisync( rtnBig:blob_t )
```

On the server side, you've got to create a blob, move the large data object into the blob, return the blob as the remote function result, and then free up the blob you've created. Here's the same code from the RPC2 *server.hla* source file:

```
procedure rtnBig;
var
    b:blob_t;
    a:array_t;

begin rtnBig;

    // Create an array to return:

    mov( @elements( array_t ), eax );
    for( mov( 0, ecx ); ecx < @elements( array_t ); inc( ecx )) do

        mov( eax, a[ecx*4] );
        dec( eax );

    endfor;

    // Copy the array into a blob:

    blob.alloc( @size( array_t ));
    mov( eax, b );
    blob.write( b, a, @size( array_t ));

    // Return the array:

    rtnBig_return( b );

    // Free the blob:

    blob.free( b );

end rtnBig;
```

On the client side, after receiving the blob (which the RPC library module has already allocated storage on the stack for), you've got to unpack the blob data into the desired data structure and then free the storage associated with the blob. Here's the client code to achieve this:

```
procedure returnArray( var a:array_t );
var
    b:blob_t;

begin returnArray;
```

```
    rtnBig();
    rtnBig_waitForReturn( b );
    blob.read( b, val a, @size( array_t ));
    blob.free( b );

end returnArray;
```