2

Hello World

This chapter demonstrates how a very simple program can be converted to a distributed application using NobleNet RPC 3.0.

Hello world program

The first example program is a version of the Hello world program that is often used to show basic programming techniques.

```
int main ()
{
	prt("Hello", "World\n");
	return (0);
}
```

```
int prt(char *text1, char *text2)
{
      printf("%s %s", text1, text2);
      return (0);
}
```

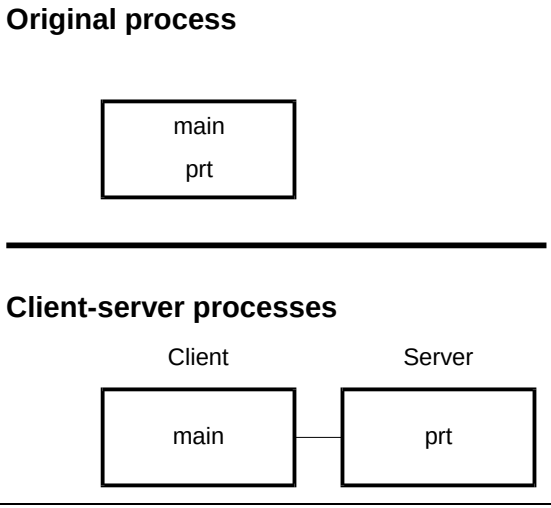
This program does nothing more than display Hello World on a screen.

Note: *The Client/Server Toolkit for C and C++ Programmers* contains a more detailed presentation of this program. This example differs in that it is aimed at the UNIX programmer.

A NobleNet RPC 3.0 Hello world program

To convert Hello world into a distributed application means that the original application must be divided into two independent processes (as illustrated in Figure 1) as follows:

Figure 1. Changing to client/server



- **Process 1**, labelled *Client* in Figure 1, runs locally. That is, it is the program that runs on the user's machine. One major function of the client program is to call the functions that execute within the second process.

  Client programs also interact with users and perform other operations that are best performed locally.

  In our examples and samples, the client program has the following UNIX file name:

  <basename>_main.c

  where <basename> is the filename of your program. (Restrictions on the length of DOS filenames requires this program name to be 3 characters or less.) An example name is hello_main.c (hw_main.c in Windows). While you can use any suffix that appeals to you, choosing something other than _main means you will also have to change the contents of the NobleNet RPC 3.0-generated *makefile*.

- **Process 2**, labelled *Server* in Figure 1, contains functions called by the client; these functions are executed on a second machine, called the server machine.

  Unless the server's name is hard-coded in the client, the user must specify the machine upon which the server is executing.

  In our examples and samples, the subroutines that are defined as RPCs are defined in a file whose name has the following pattern:

  <basename>_lib.c

  where <basename> is the basename of the program. It is also the same basename as used by Process 1. An example name is hello_lib.c (hw_lib.c in Windows).

The relationship between the client and server is simple. The client tells the server to do something (in the form of a function call), possibly sending data in the form of function parameters along with the request. The server then executes the function. Optionally, the server can return data back to the client. This cycle of sending requests and data from client to server and back can go on as long as you need it to.

NobleNet RPC 3.0 also requires that you create a file containing directives defining:

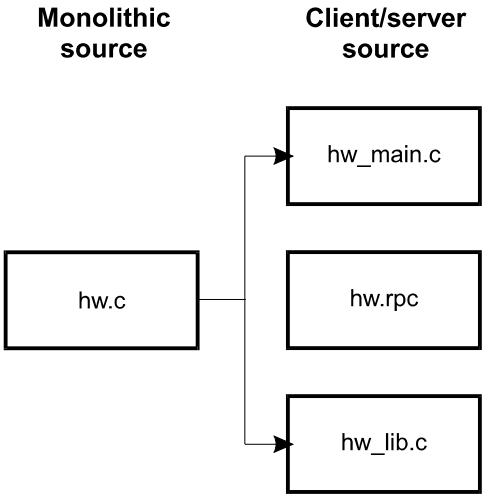
- The functions called by the client actually run on the server.
- The characteristics of the arguments sent from client to server and from server to client.

The name of this file is, by convention:

<basename>.rpc

where name is the basename of the client and server programs. An example name is hello.rpc. (The contents of this file are discussed later in this chapter.) Figure 2 shows the relationship of your original file to the three files needed when creating a client/server application.

Figure 2. Creating RPC Files



The original file, *hello.c*, is replaced by *hello_main.c* and *hello_lib.c*. *hello.rpc* is then created.

Note: For the remainder of this chapter, all filenames will use UNIX naming convention. In you are programming using Windows 3.1, your file names (that is, what we've been labelling as <basename>) are limited to three characters.

The server can call other functions without defining them as RPC functions if they are defined locally and only use information within the server process. In more complicated programs, data movement can become very sophisticated, including:

- Changing roles so that the client becomes the server and the server becomes the client.

- Calling additional servers so that one client's server becomes the client for yet another server.

hello_main.c

The hello_main.c program is the client program, and is defined as follows:

```
int main(int argc, char **argv)
{
      hello_open_transport(argv[1], 0L, 0L);
      prt("Hello", "World\n");
      return (0);
}
```

Three changes were made to this main routine:

- main now uses the standard C language argc and argv command environment variables.
- The hello_open_transport() function call was added. This function initializes RPC services. Its name is a combination of the basename and the string _open_transport(). The source for this function is generated by NobleNet RPC 3.0. See the *NobleNet RPC 3.0 Functions Reference Guide* for more information.
- The prt() function was removed. (It will be added to the hello_lib.c file.)

Unlike some RPC code generators, NobleNet RPC 3.0 lets you pass as many arguments from client to server or from server to client as your program logic dictates.

hw_main.c (Windows only)

In Windows, this program is slightly more complicated. The following program assumes that a Windows handle has already been obtained. It also does not contain any error checking.

```
HWND hwin;
HANDLE hInstance;
...
int main(int argc, char **argv)
{
      CLIENT *clnt;

      hello_rpc_init(hWin);
            hello_xdr_makeprocs(hInstance);
                  clnt = hello_open_transport(argv[1], 0L, 0L);
                        prt = (“Hello”, “World\n”)
                  hello_close_transport(clnt);
            hello_xdr_freeprocs(hInstance);
      hello_rpc_exit(hWin);

      return (0);
}
```

The differences are as follows:

- While a close_transport() call is technically always required, UNIX cleans up after a program exits better than Windows. This means that the statement is mandatory for all Windows programs and often optional for UNIX.
- All Windows programs require that a connection be made to the NobleNet-supplied RPC libraries. This is done using rpc_ init(). rpc_exit() removes this connection.
- 16-bit Windows programs require that memory be *thunked*. This is done using the xdr_makeprocs() call.

hello_lib.c

The hello_lib.c file contains the server function, and is defined as follows:

```
int prt(char *text1, char *text2)
{
      printf("%s %s",text1, text2);
      return (1);
}
```

The prt() function is simply a function. It is written in exactly the same way it would be written in a non-RPC program. Notice that no server-side main() function exists. This function will be automatically created by NobleNet RPC 3.0.

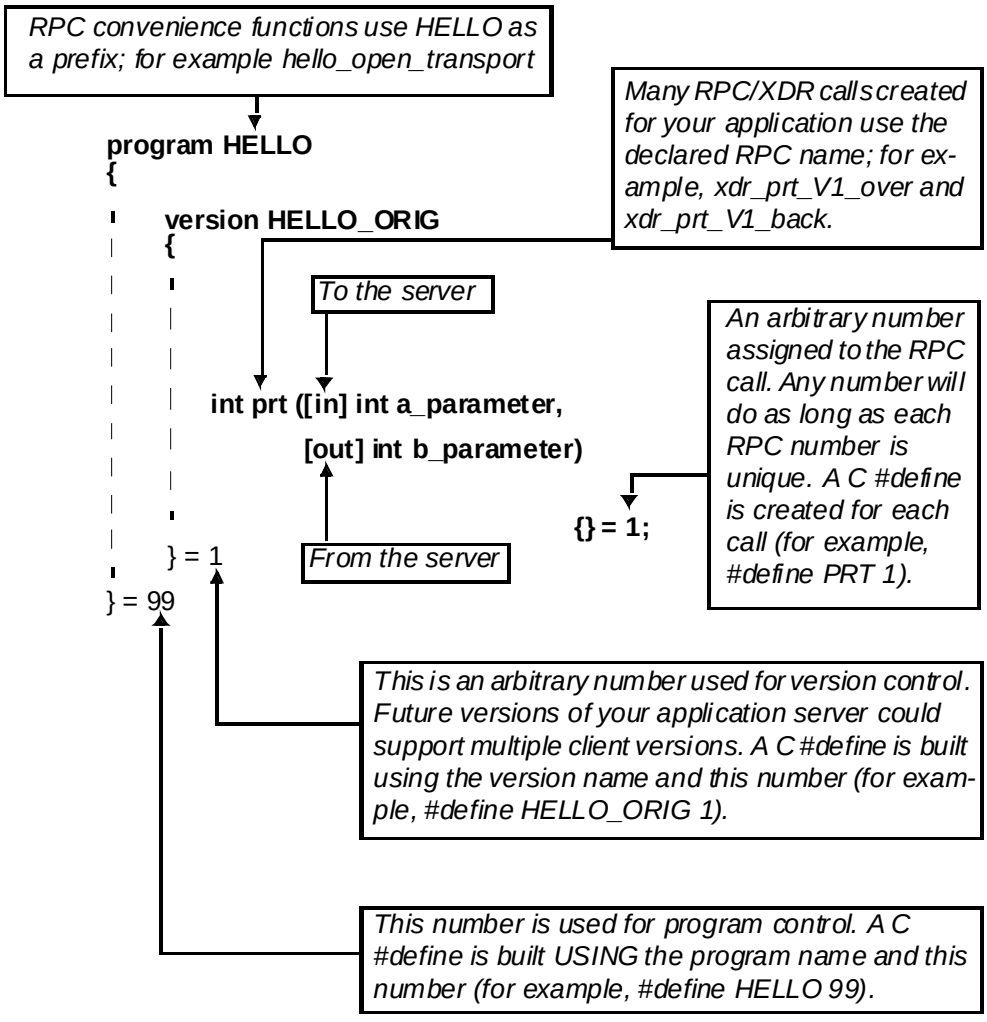
hello.rpc

Now that the program is partitioned into client and server portions, you can create the .rpc that will contain a definition of the data sent from the client to the server. In addition, NobleNet RPC 3.0 uses the information in this file to generate functions that allow the hello_main.c and hello_lib.c programs to execute separately in a client/server environment. Here is the hello.rpc file.

```
program HELLO
{
      version HELLO_ORIG
      {
            int prt([in] string8 *txt1,
                  [in] string8 *txt2) {} = 1;
      } = 1;
} = 99;
```

The language that you use to create this file is called the NobleNet RPC 3.0 Interface Definition Language (or IDL). Figure 3 contains an overview of the IDL's components.

Figure 3. IDL Structure



After NobleNet RPC 3.0 processes this IDL data, it generates the C language files that transform your program into a client/server application. (These files are described later in this chapter.)

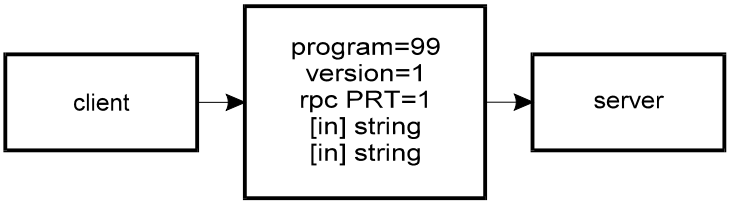
program statement

The hello.rpc file defines A program named HELLO. NobleNet RPC 3.0 transforms this name into a define constant, as follows:

```
#define HELLO 99
```

This number identifies a unique client and server protocol number that allows the client and server to find each other on a network. These unique numbers are issued by the Internet Assigned Number Authority (see page 21). The Authority also reserves numbers for use by enterprises developing applications for internal use. See Figure 4.

Figure 4. Transmitting Data



All .rpc files must contain a program keyword. (The one exception to this rule is discussed on page 51.). The string following this keyword becomes a symbolic name for the RPC program number, which is 99.

The statements within the program's curly braces following the program keyword define the program. (As you will see, some NobleNet RPC 3.0 statements can occur both before and after the program statement.) All program definitions contain one or more version statements. As this statement's name indicates, NobleNet RPC 3.0 lets you define more than one version of a program or function.

For more information on the program statement, turn to page 20.

version statement

The name following the version keyword is an arbitrary symbolic name that identifies a version of the program, and, like the name following the program statement, is used to create a define constant. In this case, NobleNet RPC 3.0 creates the following statement:

```
#define HELLO_ORIG 1
```

For more information on the version statement, turn to page 22.

Function data declarations

The curly braces following the version statement contain definitions for all server functions called by the client and the data that is sent from the client to the server and from the server to the client using these functions.

The declaration in this example defines the prt() function. (In C, all functions that do not explicitly list a return data type are declared as type int.) You should be careful when you are defining function names. Some operating systems and compilers restrict the length of certain references. NobleNet RPC 3.0 can add as many as 12 characters to this name when it generates external references.

The prt function has two arguments; each's data type is string8.[1] The in modifier indicates that the argument is an input value to the server function.

NobleNet RPC 3.0 also generates a define constant for each function name. For example:

```
#define PRT 1
```

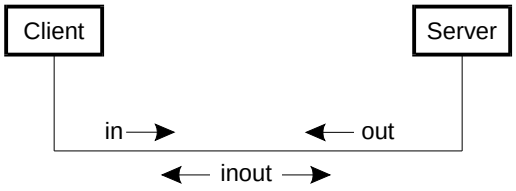
Note: Because NobleNet RPC 3.0 transforms the function's name into an uppercase define constant, all function names must contain at least one lowercase letter. If they do not, cpp (the C language preprocessor) replaces the function's name with the define constant. Because you do not see the replacement name, the error message produced by the compiler may not make much sense.

Other often used modifiers are:

- out, which represents data returned to the client from the server.
- inout, which represents data that is sent to the server, modified by the server, and returned to the client. All inout data must be pointer data.Chapter 9

1. *string8* is a special data type defined by NobleNet RPC 3.0. See Chapter 9 for more information.

Figure 5. Direction modifiers



Attributes

Immediately following the prt() function‘s closing parenthesis is a set of curly braces. These braces represent attributes that further modify this statement. The braces are required by NobleNet RPC 3.0 even if you do not use any attributes. (Attributes are presented in Chapter 9.)

Function number

Each RPC function must be tagged with an arbitrary numeric ID. In this example, this tag is the number 1. This number has no other meaning than as a symbolic identifier used by the RPC dispatcher to identify an RPC function within a program.

Running NobleNet RPC 3.0

Now that the .rpc file is created, you are ready to run NobleNet RPC 3.0 and have it generate all the files needed. Assume that you have defined the following files:

- hello.rpc
- hello_lib.c
- hello_main.c

Run NobleNet RPC 3.0 as follows:[1]

```
ezrpc hello.rpc -debug_call -DRPC_DEBUG
```

[1] The **ezrpc** command and its options are described in Chapter 10.

As NobleNet RPC 3.0 executes, it displays the following information:

```
EZ-RPC (R) V3.0: NobleNet Inc., Southboro, MA 01772.
EZ-RPC      Creating hello_rpc.h
EZ-RPC:     Creating hello_rpc2.h
EZ-RPC:     Creating hello_ez.h
EZ-RPC:     Creating hello_svr.h
EZ-RPC:     Creating hello_xdr.c
EZ-RPC:     Creating hello_clnt.c
EZ-RPC:     Creating hello_srvr.c
EZ-RPC:     Creating hello_stub.c
EZ-RPC:     Creating hello_call.c
EZ-RPC:     Creating hello_server.c
EZ-RPC:     Creating hello.mk
```

In most cases, you will want to add the indicated debug flag (which is discussed in Chapter 9) while developing your program.

The way in which NobleNet RPC 3.0 expands the .rpc file is shown in Figure 6.

Note: This example shows the files created for UNIX. Similar results occur on other operating systems.

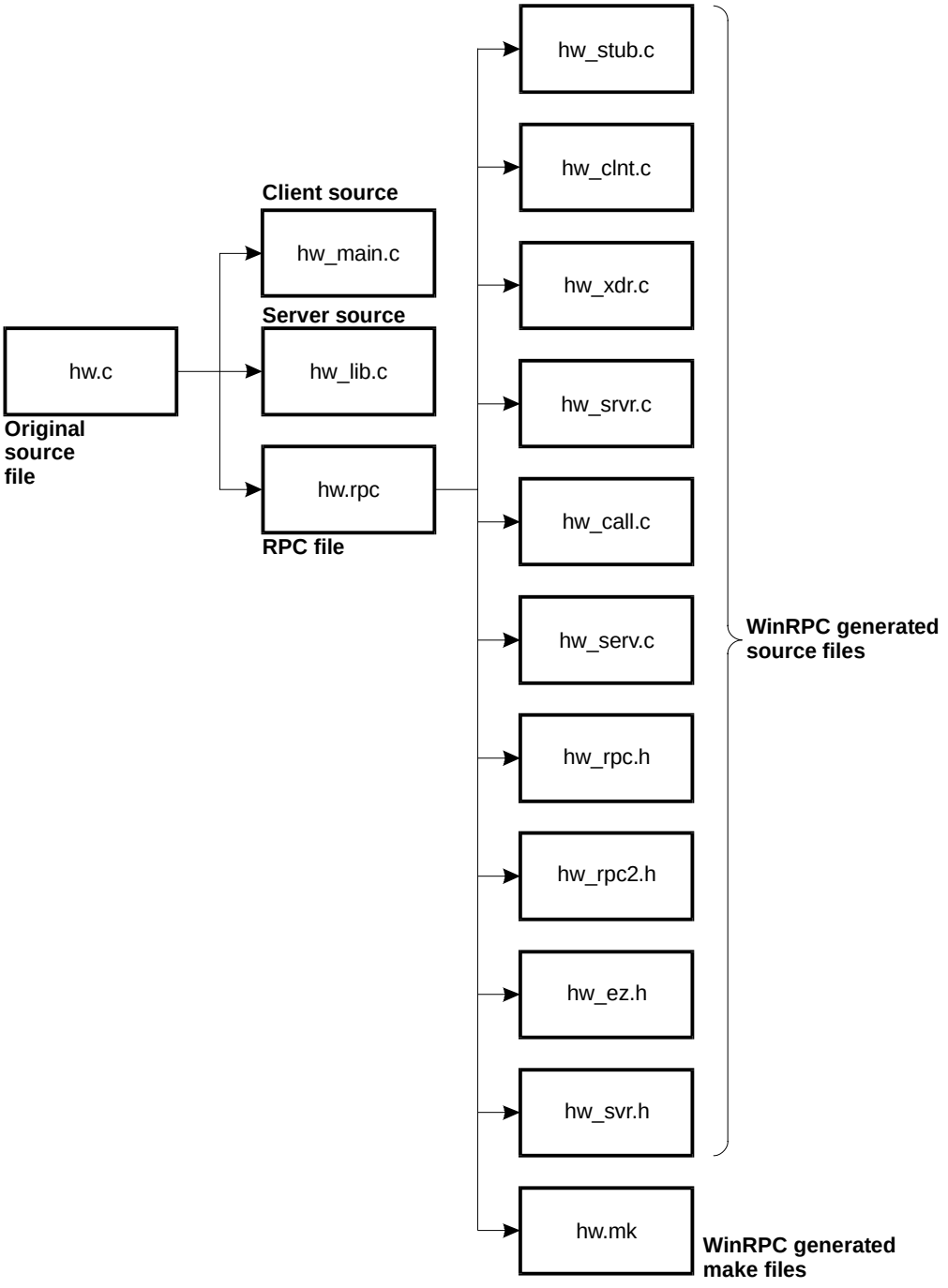
While all these files are necessary to create and run RPC programs, you need not know anything about them. NobleNet RPC 3.0 creates these source files for two reasons:

- NobleNet RPC 3.0, unlike most middleware tools, does not hide the code that it generates. Because you can see the code, you may want to tell NobleNet RPC 3.0 if what it is doing is incorrect or if you can do it better. This feature allows you to replace any generated function. (For more information, see the custom statement discussion beginning on page 103.)
- Your application will use these generated files when you make your client and server applications.

Do not modify generated files

Unlike most toolkits, NobleNet RPC 3.0 shows you its code. Because this code is automatically generated, you will lose much of the benefit

Figure 6. Changing to client/server



of using NobleNet RPC 3.0 if, after initially generating these files, you make modifications to them. As you will see, NobleNet RPC 3.0 provides all the flexibility you'll need to customize the way it generates functions and in specifying places within NobleNet RPC 3.0 generated code where your client or server functions can be called.

Creating the client and server programs

The automatically generated *makefile* contains four commonly used targets: all, client, server, and clean with all being the first target. To create the client and server processes, type:

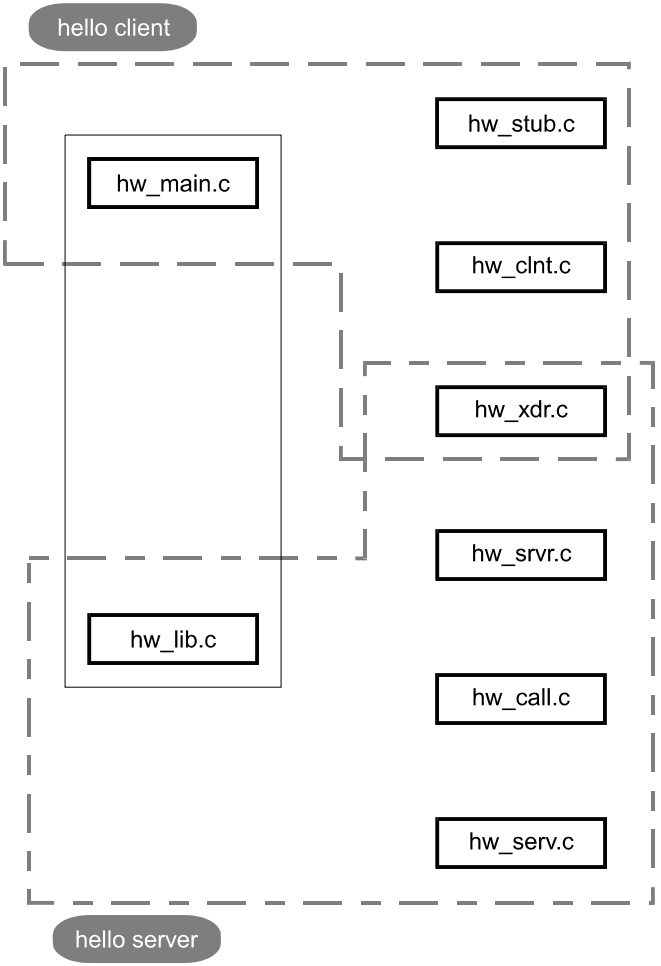
```
make -f hello.mk
```

(You would be using nmake on Windows using a Microsoft compiler, Memory Module System (MMS) on VMS, etc.) Output from the UNIX make (on an IBM RS6000) is as follows:

```
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_stub.c
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_clnt.c
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_xdr.c
ld    -r -o helloC.o \
      hello_stub.o hello_clnt.o hello_xdr.o
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_main.c
cc    -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -o hello_main \
      helloC.o hello_main.o \
      -L/home/barryk/ez30/bin/rs6000 -lezrpc
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_server.c
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_srvr.c
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_call.c
ld    -r -o helloS.o \
       hello_server.o hello_srvr.o hello_call.o hello_xdr.o
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -c hello_lib.c
cc -g  -D_BSD -D_BSD_INCLUDES -qlanglvl=ansi -o hello_server \
      helloS.o hello_lib.o \
      -L/home/barryk/ez30/bin/rs6000 -lezrpc
```

The NobleNet RPC 3.0-generated UNIX *makefile* creates client and server objects, helloC.o and helloS.o. Using these objects, NobleNet RPC 3.0 creates two UNIX executable programs called hello_main() and hello_server(). (See Figure 7.) You can now copy the server pro-

Figure 7. Making the client/server programs



gram to another machine (of the same type) and set it running. If that machine's name is *saturn*, you invoke the client program as follows:

```
hello_main saturn
```

When this statement executes, the prt function in hello_server prints Hello World on saturn. (The message is displayed in the window from which hello_server was executed.)

Killing the server

When you kill the server, you can use the kill command. However, do not use kill -9. If you use kill -9, the server's program number remains registered with the portmapper or rpcbind process. (This is discussed in more depth on page 119.)

On VMS, use stop process /id=<*process_id*> to kill the server.