NETBULA, LLC

PowerRPC Client/Server Development Tool

Power RPC Quick Reference

POWERRPC CLIENT/SERVER DEVELOPMENT

PowerRPC Quick Reference

http://www.netbula.com

Table of Contents

HOW IT WORKS **3**

PRIMITIVE TYPES **7**

POINTERS **7**

STRUCT **8**

ARRAY **8**

POINTER ARRAY **9**

TYPEDEF **10**

DISCRIMINATED UNIONS **11**

MULTI-DIMENSIONAL ARRAYS **12**

SYNTAX **13**

INTERFACE DECLARATION BY EXAMPLE **13**

RPC FUNCTION DECLARATION **15**

PROPERTY DEFINITIONS **15**

THE PROTOCOL USED BY CLIENT/SERVER FOR COMMUNICATIONS. OPTIONS ARE **16**

NO DEFAULT VALUE FOR THIS PROPERTY. **16**

FUNCTIONS IN THE GENERATED CODE **18**

- XDR FUNCTIONS 18
- CLIENT SIDE FUNCTIONS 19
- SERVER FUNCTIONS 20
- VOID PW_SERV_INIT(VOID) 22
- VOID PW_SERV_MAINLOOP(SVCXPRT*TCPSVC, SVCXPRT*UDPSVC, INT DOFORK, INT EXITIDLE) 22
- VOID PW_SERV_ONCE(STRUCT TIMEVAL *TIMEOUT) 23
- VOID PW_SERV_ASYNC_ENABLE(VOID) 23
- VOID PW_SERV_ASYNC_DISABLE(VOID) 23
- VOID PW_FREE_REFERENCE(XDRPROC_T XDRFUNC, VOID* PTR, U_INT SIZE) 23
- VOID PW_SERIALIZE(FILE*FP, XDRPROC_T XDRFUNC, VOID* PDATA, ENUM XDR_OP OP) 24

A FILE SERVER **25**

- DESIGN OF INTERFACE 25
- SERVER IMPLEMENTATION 27
- THE CLIENT 28

A TALK PROGRAM **28**

- DESIGN OF THE INTERFACE 29
- IMPLEMENTATION 29
- USAGE 31

ASYNCHRONOUS RPC **32**

OTHER SAMPLE PROGRAMS **34**

INDEX

Chapter 1

Introduction

Remote Procedure Call (RPC) is a powerful mechanism for client/server and distributed computing in general.

PowerRPC is a rapid RPC development tool that helps unleash the full power of RPC programming. It makes arbitrary C functions callable across process boundaries over a TCP/IP network, and it is very easy to learn and use.

PowerRPC mainly consists of the powerRPC IDL compiler and the powerRPC runtime time library. Given an interface description in a form almost identical to C function declarations, the IDL compiler generates client and server stub code - the code handles the networking and server dispatching. Unlike the original ONC RPC, PowerRPC handles generic multi-argument C functions. The arguments of the C functions can be of IN, OUT or INOUT directions, and they can be any complex C data types such as structs, linked list, unions, multi-dimensional arrays, etc. The problem of transporting data between different architectures are solved by the XDR mechanism, this way, RPC clients and servers live in any platform.

Since one of the main design goals of powerRPC is to bring the power of RPC programming to general C/C++ programmers, it is made very easy to use. In powerRPC, the programmer is insulated from network programming at TCP/IP level and the ONC RPC level. The programmer only needs to provide an interface description and their server implementation. A programmer without any prior experience of network and RPC programming experience can write sophisticated, robust and efficient powerRPC programs in a very short time.

PowerRPC is also designed to provide elegant solutions to common problems in client/server computing. For example, it allows you to create multi-tasking or multi-threading servers, it supports asynchronous server... and you can enable these features simply by specifying properties in the interface description.

How it works

One of the main components of powerRPC is its IDL compiler. Given an interface definition, powerRPC generates the RPC stub code for network

transport and server dispatch. An interface definition language (IDL) file declares the C functions provided by an RPC interface and the data types used by these functions as arguments or return values. An example of a the IDL file looks like the **quote** RPC interface shown below, whose purpose is to allow a client (the RPC caller) to get the stock quotes for a given ticker symbol,

```
% cat quote.idl

typedef struct {
        char   Ticker[8];
        double Low;
        double High;
        double Close;
} stkQuote;

interface quote {
    int  getQuote(inout stkQuote * pQuote);
} 0x12345;
```

There is nothing special about the typedef--it is pure C. The powerRPC keyword ***interface*** followed by identifier quote starts the declaration of an RPC interface, which encloses a set of function declarations in curly braces. Here we have only one function, **getQuote(),** which takes a pointer to **stkQuote**. The ***inout*** keyword specifies the direction of the argument **pQuote**. In this case, the argument is used for both input and output - the client sends the struct over to the server, the server uses the Ticker field to find quotes, and sends them back. The integer 0x12345 is just an identification number for this RPC interface, which maps to the program number in ONC RPC.

We compile the interface with the command,

```
% powerRPC quote.idl
```

Six files will be generated,

Filename	Purpose
quote.h	Header to be included by the client and server.
quote_svc.c	Server stub, it calls the server implementation of getQuote().
quote_imp.c	Template for the server implementation, programmer needs to fill in the details.
quote_cln.c	Client stub, it defines the getQuote() function for the client, this function is the interface to the server's getQuote() function.
quote_xdr.c	XDR routines used by both the client and server to encode and decode the arguments and return values.
quote.mak	A makefile template.

Our lucky programmer now needs to provide the **getQuote()** function for the server, the IDL compiler generates a file named **quote_impl.c** which contains the template for the server code. We now edit this file and add in the details, it look like this,

```
    %cat quote_impl.c

    #include "quote.h"

    int getQuote( stkQuote * quote) {

         /* let's just return some good value. */

          quote->Low = 30;
          quote->High = 35;
          quote->Close = 34;
          return 0;
     }
```

Now, we can compile and run the server ,

```
        % cc -o quoteserv quote_svc.c quote_xdr.c quote_impl.c -
lpwrpc
        % quoteserv
```

The server starts up and announces its registration with the portmapper (or rpcbind on SYSVR4). Now the **getQuote()** function is "exposed" to the network and is callable by an authenticated client.

We now write a simple client program, in file *quote_call.c*,

```
  % cat quote_call.c
```

```
  #include "quote.h"
  int main(int argc, char**argv)
  {
    stkQuote q;
    if(quote_bind(argv[1], 0,0,0)==NULL)
    {
       printf("Fail connect to server on host %s\n", argv[1]);
       exit(1);
    }
    strcpy(q.Ticker, argv[2]);

    /* make the RPC call */
    if(getQuote(&q) <0)
    {
       printf("Fail get quote for %s\n", q.Ticker);
    }
    else {
      printf("Quote for %s (LOW, HIGH, CLOSE):\n", q.Tikcer);
```

```
      printf("              %f  %f  %f\n", q.Low, q.High, q.Close);
    }
    quote_unbind(0);
  }
```

We used three functions produced by powerRPC, **quote_bind()** binds the RPC connection to the server, **quote_unbind()** close the RPC connection, and **getQuote()** calls the RPC function.

We compile the quote client,

```
% cc -o quoteclnt quote_cln.c quote_xdr.c quote_call.c
```

Suppose quoteserv is running on host eagle, we run the client as follows

```
% quoteclnt eagle <ANY TICKER>
```

You will see the set of values printed on client's terminal.

Let's summarize the steps in powerRPC programming,

1. Create an interface definition (the IDL file), which declares RPC functions and related types.
2. Run powerRPC to generate client/server stubs from the IDL file
3. Code the server implementation of the RPC functions declared in the interface.
4. Code the client, which calls the RPC functions. With the three steps above, the server is complete. Before making RPC calls, a client needs just one additional step: call the **<interface>_bind()** function to bind to a server.

Starting from next chapter, we will learn more about powerRPC IDL.
Next: Type declarations in the

Chapter 2

Type declarations in the powerRPC IDL

PowerRPC IDL closely resembles the C function declaration.

The powerRPC IDL is an extension of the C type and function declarations. The IDL is made as close to C as possible, therefore almost every valid C declaration is also a valid powerRPC declaration, with a few exceptions. Function pointer is such an exception, it is an error to declare a pointer to function as a member of struct or union in powerRPC, for obvious reasons.

Primitive types

You can use any primitive types, such as

char, short, int, long, float, double, enum

in the IDL. When applicable, you can also use the ***unsigned*** specifier before the types.

However, currently, powerRPC does not support non-standard types such as *long long*, *long double*. Anyway, you should avoid using such types for portability.

Pointers

In powerRPC IDL, a declaration of the form

```
T * ptr;
```

is always taken as a pointer to a single object of type **T**. To declare “pointer arrays", please refer to page 8.

Example:

```
struct link_t {
    struct link_t * next;
    double val;
};
```

struct link_t can be used as a singly linked list of **double**s.

The powerRPC generated XDR function will recursively chase the pointers when encoding/decoding an object. For the above declaration, when next is not NULL, the generated function **xdr_link_t()** will be called for ***next**. This implies that we can not use self-referencing types such as double linked lists as RPC arguments, since the pointers form a circle, which would cause infinite recursion in the XDR function.

When use a pointer as an RPC argument, you must make sure that it is properly initialized. Otherwise, either a memory fault or a memory leak may occur. Suppose you have an RPC function **foo(out int * ptr)** (where the ***out*** keyword declares that the argument is used to receive result from server), the following lines of code

```
int * pi; // not initialized
foo(pi);
```

will probably cause a fault since a dangling reference is passed, while this line of code

```
foo(0); // passed a NUL pointer
```

would results in a memory leak, even if the server implementation of **foo()** does not use the ptr argument at all.

struct

structs are very important in C, since it is the only way to support aggregated types. PowerRPC IDL allows structs that consists of data fields of any types, such as primitive types, **typedef**s, pointers, arrays, **struct**s, etc.

Array

In making an RPC, the function arguments are sent to the server, which resides in different address space, usually on a remote host. We want to minimize the data transfered to reduce the overhead. When the data to be sent is an array, we should only send those needed elements. In powerRPC, when an argument is declared as **char msg[1024]** with a constant 1024 as the array length, exactly 1024 characters will be sent to/from the server. To tell powerRPC to be more efficient, we can define the size attribute for a fixed array, as shown below,

```
struct msg_t
{
   int len;
   char msg[size=len, 1024];
};
```

where we specify that the number of needed elements in array msg is len.

Or even this,

```
struct msg_t2
{
   char msg[size = strlen(msg)+1, 1024];
};
```

The size attribute is used only during the code generation of the XDR routines, the generated header file contains the C declaration with the **size = *expression*** part stripped out.

One important restriction on the expression used for the size attribute: one can not use global variables. This is because the server and client use the XDR routines symmetrically, a variable in one address space is different from one in another address space.

PowerRPC does not recognize declaration of extern variables or functions, and it does not check the type of the expression used for the size attribute.

pointer array

The C declaration of a pointer is ambiguous. A **char *** means (at least) two different things,

1. a pointer to a single char,
2. a pointer to the first element of an array.

A C programmer, had decided the meaning in his mind when he wrote the code. However, powerRPC (or a maintainer of other people's code too?) is not able to guess this implicit meaning. The C programmer usually writes a comment such as "this is a null terminated string", so the reader of his code knows what a char* really means. We must do the same thing for powerRPC.

In powerRPC IDL, a simple **T*** will always be interpreted as a pointer to a single object of type **T**. To declare a ``pointer array'', you must used the following syntax,

T [**size**=*expression*, **maxsize**=*expression*] ptr;

Notice the position of *`[size_attrib]* is before the identifier **ptr**. The C declaration corresponding to the above is simply **T *ptr**, obtained by replacing the [*size_attrib*] before the identifier **ptr** with a *.

We have mentioned about the size attributes for fixed arrays. For ``pointer array'', we should have an additional attribute, maxsize to specify the maximum number of elements, using the **maxsize = *expression*** syntax within the subscript operator **[ ]**. You may also specify the maximum size with a integer constant, without using the **maxsize = *expression*** syntax.

Example:

```
struct msg_t2 {
       int  maxlen;
       char [size = (msg? strlen(msg)+1 : 0), maxsize=maxlen] msg;
       char [256] msg2;
       char msg3[256];
};
```

Notice that we did not specify the size attribute for msg2, in this case powerRPC will take the maxsize, which is 256, as the size. However, you must specify at least one of the sizes for an array, otherwise an error will be reported by powerRPC.

The C declaration of the above is

```
     struct msg t2 {
         int maxlen;
         char * msg;
         char * msg2;
         char msg3[256];
     };
```

It is your responsibility of to make sure that msg and msg2 are initialized and pointing to allocated memory before using it as RPC arguments. You must be very careful, or a memory fault will occur, for example, the declaration claims member msg2 is an array of 256 elements, so powerRPC will try to faithfully deliver 256 chars. If msg2 were a NULL pointer, a fault would be inevitable.

Typedef

Typedefs are very useful in powerRPC. A generated XDR function for type T is always of the following prototype,

```
int xdr_T(XDR*, T* pT);
```

i.e., it takes only two arguments, the first is a pointer to an XDR object, and the second is the pointer to the object(of type T) to be marshalled. To encapsulate the size information of an array in an XDR function, we need to use typedef.

For example, we can define an array of 512 chars,

```
typedef char c_arr512[512];
```

To define a C string (char*) with a maximum length of 1023,

```
typedef char [size = strlen(*this)+1, 1024] str1024;
```

here we used the powerRPC keyword **this**, which can be used in a type declaration (typedef, struct or union) to signify the pointer pT passed to the XDR function of the typedef.

The corresponding C declaration for **str1024** is simply **char ***, however, the elaborate size specification above actually defines str1024 as a C string. In

powerRPC, instead of being provided a fixed number of predefined types such as **string8** (meaning C string), you can use its expressive IDL to defined your own types. Suppose you need to declare an array of ***long*** integers ending with a zero, you can do it yourself as follows,

```
typedef long [size =strlen32(*this)+1, 1024] string32;
```

provided that you define the **strlen32()** function somewhere like this:

```
int strlen32(long*la) {
     int i;
     for(i=0; la[i]; i++);
     return i;
 }
```

Discriminated unions

Union is a space saver construct.

A discriminated union must be declared in a struct, and must use an integral expression as the discriminator. The C switch statement syntax is used to select from the choices, as shown in this example,

```
struct primitive_t
{
     char choice;
     union switch( choice )
     {
      case 'i': int ival;
      case 'c': char cval;
      case 'd': double dval;
     } value;
};
```

A discriminated union must be defined without a tag name, to prevent it from being used outside of the struct.

The corresponding C declaration of the above is,

```
 struct primitive_t
{
     char choice;
     union
     {
           int ival;
           char cval;
           double dval;
     } value;
};
```

To use the type **primitive_t**, you must assign the choice field, and the corresponding union member, as shown in the following example,

```
struct primitive t aprim;
/* we are using it as a double */
aprim.choice = 'd';
```

```
        aprim.value.dval   = 9.9 ;

        /* now we can use aprim in an RPC */
```

Sometime we may want to use a union to represent optional data. To do this, we simply set the discriminator to a case not listed in the ``switch statement''. Thus, if we set the choice to a undefined case,

```
        aprim.choice = -1;
```

No data will be transferred when aprim is later used in an RPC argument.
A usual C union declaration is also allowed. However, it will be treated as opaque data in powerRPC, that is the raw bytes (non-portable) of the union will be transferred across the network.

Multi-dimensional arrays

Multi-dimensional arrays or pointer arrays can be used in powerRPC, however, only the size of the first dimension can be a variable. To use multi-dimensional arrays with variable sizes at second dimension and above, you can use typedefs.

For example, we can use the **str1024** to define an array of strings.

```
      struct string array
      {
             int len;
             str1024 [size=len] strArray;
      };
```

Next: Interface declarations in the

Chapter

3

Interface declarations in the powerRPC IDL

We have seen a simple example of the interface declaration in Chapter 1. Now that we have learned how to declare types, what remains is simple: to declare RPC functions, using the declared types in function signatures or as return types.

Syntax

A powerRPC IDL files may contain the following components,

- C preprocessor pseudo-ops, such as #include. PowerRPC will pass the IDL file through CPP before performing its own parsing. It also defines a macro **POWERRPC_COMPILE** by itself when invoking **CPP**.
- Type declarations, as described in chapter 1.
- A single interface declaration that specifies the RPC functions. This must appear after all type declarations.
- Copy-out statements. These are arbitrary text followed by a leading **%**. They are copied AS IS to the header file generated by powerRPC. Those appear before the interface declaration are prepended to the header file, and those appear after are appended to the header file.

Interface declaration by example

An interface declaration consists of property definitions and functions declarations.

The following is an example,

```
1  interface test {
2     property TRANSPORT_PROTOCOL = tcp;
3     int my_read(
4         out char [maxsize=maxlen, size=return>0?return:0] buf,
5             int maxlen
6         )
7        {
8            property FORK_ON_CALL = true;
10       };
12    int my_write(in char [size = len] buf, int len);
13 } = 12221;
```

At line 1, the keyword **interface** followed by the identifier test announces our RPC. At line 2, we set the TRANSPORT PROTOCOL of the RPC to be TCP.

Properties are optional settings that customize RPCes on the client or server side. If the TRANSPORT_PROTOCOL property is not set, the default is tcp_and_udp, meaning the client can connect to the server via both protocols.

From line 3 to line 10, we declared an RPC **function my_read(char* buf, int maxlen)**, the **out** keyword on line 4 says that the **buf** is used for output only, and its maximum size is determined by the second argument **maxlen**, its size is determined by the expression **(return > 0? return:0)**, where the **return** keyword is the return value of the RPC function.

Obviously, **return** can ONLY be used in size expressions for ***out*** arguments. The FORK_ON_CALL property says that the RPC server will fork a child upon receiving the **my_read()** call, so the parent can handle other requests.

Line 12 declares another RPC function **int my_write(char* buf, int len)**, in this case, the **buf** is used for input only, the size of **buf** is determined by the **len** argument. **my_write()** does not have additional properties. While my_read() function has set the **FORK_ON_CALL** property, the server will not fork a child to handle it.

Line 13 specifies that the test RPC is identified by the program number **12221**. The program number is a long integer used to identify the RPC program, and it should not be in conflict with other RPC programs.

PowerRPC IDL allows you to specify the direction of the argument as one of **in**, **out** and **inout**, the default direction is **in**. You can use an argument in the size expression for an array argument.

The property definitions have scopes. When a property is defined in the scope of the interface, it is shared by all RPC functions. However, a function can redefines a particular property, and overrides the common value.

RPC properties are very useful in customizing how RPC works. They can be a boolean that toggle between options, or a value to set a parameter, or a function to be called at a particular point.

RPC function declaration

RPC function declarations should be enclosed in the body of the interface declaration. A function can use any of the types declared previously in the IDL file. An array argument can use an argument of the same function in its size expressions. Thus, we can have the following,

```
interface hello {
        void printmsg(char [size=strlen(msg)+1, 1024] msg);
} 0x12345;
```

A function argument can be of one of the three directions,

in

The argument is only to be sent to the server. This is the default (when no direction is specified for an argument).

out

The argument is used only to receive result from server, it must be a pointer or an array.

inout

The argument is used for both sending and receiving data, it must be a pointer or an array.

The return type of an RPC function can also be of any type. However, you must note that for a functions that return a pointer, powerRPC allocates the memory to which the pointer points, and it is the programmer responsibility to free that memory using the library function **pw_free_reference**().

Property definitions

As shown in the previous examples, property definitions come in two places: immediately enclosed in the "body" of an interface declaration, or inside the "body" of a function declaration.

The following properties can be defined.

VERSION

An integer value to specified version number for the interface. The default value is 1.

TRANSPORT_PROTOCOL

The protocol used by client/server for communications. Options are

tcp

A connection-oriented reliable protocol, with exactly once calling semantics.

udp

A connectionless and unreliable protocol. The client would retransmit the RPC call if not receive response within a timeout period. Another restriction is that UDP datagrams usually have a maximum size of 8K bytes, so it may not be suitable for RPC functions whose arguments or return value are large.

tcp_and_udp

The server will register with both protocols. The client can choose either one of them, with the default being TCP.

The default is **tcp_and_udp**.

SERVER_PORT

An integer value used to specify to the port number of the RPC server. When this property is not set (the usual case), the server choose an arbitrary port that is available and the client consults the portmapper on the server host to obtain the port number. When this set, the client will bypass the portmapper and uses the port number specified.

NO_PMAP_REGISTER

When this property is set to **true**, the server will skip registration with the portmapper. The rpcinfo command won't find your server. You would normally set this property when you also specified a fixed port.

The default is false.

INIT_BEFORE_REGISTER

A user defined function to be called before server register itself. This function must be of the type of void (*) (int, char**), it is passed the argc and argv arguments from the main(int argc, char**argv) function.

No default value for this property.

INIT_AFTER_REGISTER

The user defined function to be called after server register itself. This function will be passed the argc and argv arguments from the main(int argc, char**argv) function.

For example, you could use this initialization function to set up signal handlers, a useful one is to unregister your RPC server with the portmapper when a SIGINT is received. Using the quote RPC server as an example, we can write the following functions,

```
void handler(int sig)
{
        quote_1_unmap(0,0);
        exit(0);
}

void set_sigint_handler(int argc, char**argv)
{
      signal(SIGINT, handler);
}
```

where quote_1_unmap() is a function generated by powerRPC for the purpose of unsetting the portmapper entry of the quote server. By setting

```
INIT_AFTER_REGISTER = set_sigint_handler;
```

you make sure that the quote server will remove its entry from the portmapper's database.

No default value for this property.

GEN_MAIN_FUNC

When set to false, the powerRPC compiler will not generate the main() function for the server.
The default is true.

FORK_ON_CONNECTION

When set to true, the RPC server will fork a dedicated server for each client, all RPCes from the client will be handled by the child. For TCP, the child is created when a request for connection is received, when the client detaches from the server, the forked child will exit. The semantics with UDP transport is different, since UDP is not connection oriented. For UDP, the server would fork to handle each incoming RPC call. It is recommended not to use this option with UDP transport, use FORK_ON_CALL instead.

The default is false.

FORK_ON_CALL

When set to true for an RPC function, the server will fork a child to handle the call, so itself can still handle other requests.

The forked child will exit at the completion of the call.
The default is false.

NON_BLOCKING

When this is set to true for an RPC function, the server will reply to the server immediately, before it actually calls the user implementation of the function.
Obviously, the return type of such an RPC function should be **void** and it should have no **out** or **inout** arguments.

TIMEOUT_VALUE

This is an integral value for the RPC timeout in seconds on the client side. When specified for an RPC function, a timeout error will occur when a reply for this function is not received within the timeout.
For example, if you set this to be 0 for function foo(), then every call of foo() will timeout.
The default value is 60.

SERV_CALL_PREFIX

For an RPC function foo() , the powerRPC compiler will generate the client definition of foo() for you, and you must write the server implementation of foo() yourself. However, sometimes, you want to make a program both a server and a client of the same RPC interface. To make this possible, you must set this property.
For example, when you set

```
property SERV_CALL_PREFIX = serv_ ;
```

PowerRPC expects your sever implementation of foo() to be serv_foo(). This solves the name conflict and you can write a program that is both a server and also a client (of another server) of the same RPC.

Functions in the generated code

Various functions are generated from your IDL file, occasionally, you may want to call them in your code directly.

XDR functions

For any data type **T** used in an RPC function, an XDR function **`bool_t xdr_T(XDR* xdrs, T*ptr)`** is generated when **T** is not one of the primitive types. This function takes a pointer to an object of type **T** as the second

argument, and performs encode, decode or free operation based on the value of `xdrs->x_op`.
One possible use of the XDR functions is for serializing data to and from files, this allows one to save complex data structures in a portable binary format.

Client side functions

CLIENT* <*interface*>_bind (char*host, u_long pno, u_long vno, char*protocol)
Parameters:

host

the host name of the RPC server

pno

the RPC program number. When this is 0, the one specified in the interface is used.

vno

the RPC version number. When this is 0, the one specified in the interface is used.

protocol

``tcp" or ``udp". When this is 0, the one specified in the interface is used.

Return value: On success, it returns a pointer to CLIENT struct. On failure, it returns NULL.

This function binds the client to the RPC server specified by the parameters. All subsequent RPC calls go to the specified server. The client application may save the returned client handle, so it can talk to multiple servers.

void <*interface*>**_bind_handle (CLIENT*pclnt)**

Parameters:

pclnt

a valid client handle returned from a previous call to <*interface*>_bind().

This function binds the client to a previously bound server. Subsequent RPC calls go to this server.

void <*interface*>**_unbind (CLIENT* pclnt)**

Parameters:

pclnt

a client handle returned from a previous call to **<interface>_bind().** If this is NULL, defaults to the current client handle.

This function unbinds the client to a server.

enum clnt_stat <*interface*>**_errno (CLIENT* pclnt**)

Parameters:

pclnt

a client handle returned from a previous call to <interface>_bind(). If this is NULL, defaults to the current client handle.

This function returns the RPC call status after an RPC has been made. If this is not RPC_SUCCESS, an error condition has occured.

Server functions

SVCXPRT*
<*interface*>_<*version*>**_reg(int sock, u_long pno, u_long vno, int protocol)**

Parameters:

sock

a bound socket or *RPC_ANYSOCK*.

pno

the RPC program number. When this is 0, the one specified in the interface is used.

vno

the RPC version number. When this is 0, the one specified in the interface is used.

protocol

IPPROTO_TCP or *IPPROTO_UDP*.

Return value:
A pointer to the server transport. NULL on failure.
This function takes register the <interface> RPC server with the specified parameters.

void <*interface*>_<*version*>**_main(int argc, char**argv)**

Parameters:

argc

Number of command line arguments.

argv

The array of command line arguments.

This function starts the RPC server. It should never return.

void <*interface*>_unmap (u_long pno, u_long vno)
Parameters:

pno

the RPC program number. When this is 0, the one specified in the interface is used.

vno

the RPC version number. When this is 0, the one specified in the interface is used.

This function undoes the RPC server registration with the portmapper.

Next: PowerRPC library functions

Chapter

4

PowerRPC library functions

PowerRPC comes with a runtime library that provides much of the high level RPC functionality. Here we list the ones that you may want to know about.

void pw_serv_init(void)

A server must call this function first to do proper initialization of RPC runtime library.

void pw_serv_mainloop(SVCXPRT*tcpsvc, SVCXPRT*udpsvc, int dofork, int exitidle)

Parameters:

tcpsvc

A TCP server transport returned from a call to server register function.

udpsvc

A TCP server transport returned from a call to server register function.

dofork

If true, then the server would fork a dedicated child server for each client.

exitidle

Should always be 1.

This function starts the server. It never returns.

void pw_serv_once(struct timeval *timeout)

Parameters:

timeout

A time out value.

Ask the server which is not started by *pw_serv_mainloop()* to serve pending RPC calls. If there is no pending calls, it waits for calls during the time period specified by *timeout* parameter.
The call returns after pending calls have been done, or timeout expires.

void pw_serv_async_enable(void)

Enables an asynchronous server. Instead of waiting for requests, the server is signaled when calls arrive.

void pw_serv_async_disable(void)

Disable the asynchronous server previous enabled.

void pw_free_reference(xdrproc_t xdrfunc, void* ptr, u_int size)

Parameters:

xdrfunc

An XDR function.

ptr

A pointer to memory allocated by the XDR function.

size

The size of the memory block.

Sometime, the underlying RPC mechanism allocates memory, a typical example is when an RPC function return a pointer. To free this memory, one needs to call this function. As in the following code segment,

```
 T * pT;
pT  = foo() ; /* foo() is an RPC */

...

/*free it */
pw_free_reference(xdr_T, pT, sizeof(*pT);
```

void pw_serialize(FILE*fp, xdrproc_t xdrfunc, void* pdata, enum xdr_op op)

Parameters:

fp

A opened FILE pointer.

xdrfunc

An XDR function.

pdata

A pointer to data.

op

XDR operation, can be XDR_ENCODE or XDR_DECODE.

This function can be used to serialize/deserialize a C data structure to a file in a platform independent format. When *op* is XDR_ENCODE data is serialized to the file, when *op* is XDR_DECODE data is read from the file and the data structure is reconstructed in memory.

Next: Tutorials

Chapter

5

Tutorials

Learn by example

In this chapter, we will go through some sample powerRPC programs.

A file server

In this tutorial, we show you how to write a file server, and then write a client that understands some simple commands similar to those in FTP, such as GET, PUT, LS and CD. Since you are completely relieved from the task of writing networking code by using powerRPC, this assignment become rather trivial - once you get familiar with how RPC works.

Design of interface

Our file server should provide a set of RPC functions that allow a client to access files located at the server machine. The server would open the file on client's behalf, and subsequently, read from or write to the file when such operations are requested from the client.

It will be convenient to make our RPC functions look like existing file I/O functions. For example, we could make our RPC interface resembles the UNIX file I/O system calls, read(), write(), and chdir(), etc. For our demo, let's mimic the C stdio library functions, such as fread() and fwrite(). To distinguish our RPC from the C library functions, we prefix our functions with a lower case 'r', so our RPCes will be rfread(), rfwrite(), etc , we will define a type called rFILE corresponding to the FILE type. So the rfread() function is of the following prototype,

```
int rfread(void * buf, int size, int nmemb, rFILE* stream);
```

of course, we should also have a rfopen() function to open a remote file.

Assuming an RPC connection has been established, a client could read from a file "foo" sitting on the server's machine like this,

```
rFILE * fp;
char    buf[1024];
fp = rfopen("foo", "r");
rfread(buf, 1, 1024, fp);
```

Having decided the interface functions, we need to decide the transport protocol. UDP is unreliable and the sizes of datagrams are usually limited to 8K, so let's choose TCP.

Another decision we need to make is the statefulness of the server. A stateful server maintains the state for the client, whereas in a stateless server, the client supplies all the information (such as current file offset) at every RPC. In our demo, we choose to use a stateful implementation.

The stateful server works as follows. The server maintains a table of opened files, when the client makes a rfopen() call, the server uses fopen() to open the file and record the FILE* pointer in the table, the index to that table entry is return to the client. The client then use the index to reference the file when it makes rfread(), rfwrite() and rfclose() calls later. Obviously, This index must be a field of our rFILE structure.

To implement our client program (simple FTP), we also need an RPC to list the contents of the remote directory. Thus we have the rlistdir() function, which returns a linked list of directory entries in its argument.

Combining these ideas we come up with the following interface declaration,

```
typedef  char [size=strlen(*this)+1, 1024]  str1024;
typedef  char  c_arr1024 [size=strlen(*this)+1, 1024];
typedef  unsigned long size_T;

typedef struct fHandle {
       int fd; // identify the file on server
}rFILE;

typedef struct dentry {
               c_arr1024      name;
               struct dentry *next;
}DENTRY;

interface rfile {
     property TRANSPORT_PROTOCOL= tcp;
     property INIT_BEFORE_REGISTER= init_fdtable;

     rFILE* rfopen(in const str1024 filename,
                   in const str1024 mode)
             {
                  property     TIMEOUT_VALUE = 2;
```

```
            };

     int    rfread
            (out void [maxsize=size*nm, size=return>0?return:0] ptr,
             size_T size, size_T nm, in rFILE* stream
             );
     int    rfwrite
            (in const void [maxsize=size*nm, size=nm*size] ptr,
             size_T size, size_T nm, in rFILE* stream);
     int    rfclose(rFILE*stream) ;
     int    rlistdir(in str1024 path, out DENTRY * pent);
     int    rchdir(in str1024 path) ;
} 0x5555 ;
```

Note that we defined the property **INIT_BEFORE_REGISTER**, which is the function to initialize the table of FILE* pointers to 0 on the server.

Server implementation

You need to write the server implementation of the rfopen(), rfread(), etc. The rfopen() functions merely fopen()s the file, and record the FILE* pointer in the global fd_table and return the index to the client.

The rfread() function is listed below,

```
FILE * fd_table[MAXFILE];

int   rfread(void *ptr, int size, int nm, rFILE * stream)
{
    FILE           *fp;
    int             index = stream->fd;

    /* first check if the index is valid,
       if true, get the FILE pointer */
    if (index < 0 || index >= MAXFILE
        || (fp = fd_table[index]) == 0)
    {
        fprintf(stderr, "Invalid rFILE pointer!\n");
        return -1;
    }
    return fread(ptr, size, nm, fp);
}
```

That is it! The rfwrirte() function is defined by replacing the word read in the above with write.

The rchdir() function is even simpler.

```
int   rchdir(char *path)
{
    return chdir(path);
}
```

The rlistdir() function is a bit more complicated, but probaly you can make it simpler by writing a more elegant linked list.

The client

Now the server code is complete. When it is compiled and executed, it makes the six RPC functions to be callable from anywhere in a network. A programmer can make use of these functions and write whatever applications he/she wants. Given the available **RPC** functions, we can easily write a file transfer client program which supports **GET, PUT, LS** and **CD** commands.

The **get_file()** function, which reads a remote file **src** and saves it in file **local**, is listed below,

```
int   get_file(char *src, char *local)
{
    rFILE          *fp = 0;
    FILE           *localfp;
    char            buf[1024];
    int             cnt;

    localfp = fopen(local, "w");
    if (!localfp) {
        perror(local);
        return -1;
    }
    fp = rfopen(src, "r");

    if (fp == 0) {
        fprintf(stderr, "Fail to open remote file!\n");
        fclose(localfp);
        return -1;
    }
    while ((cnt = rfread(buf, 1, 1024, fp)) > 0) {
        fwrite(buf, 1, cnt, localfp);
    }

    fclose(localfp);
    rfclose(fp);
    free(fp);
    return 0;
}
```

The code above is almost exactly what one would do to copy one local file to another using the **fread(), fwrite()** functions. The only difference is in the **free(fp)** call. PowerRPC always allocates the memory for a return value of reference type, it is the caller's responsibility to free that memory.

A talk program

In this example, we will write a talk program that allow two users to send messages to each other.

Design of the interface

A true talk program must have some daemon to notify a user that someone else is trying to initiate talk with him/her. We can certainly write such as server using powerRPC, however, this is not the purpose of this demo, since you can already write such a server after learning the powerRPC from the material above.

Our talk system will be just one program named talk2. One user starts talk2 first, and another user executes the same program to communicate with the first user, knowing he/she is there waiting. Talk2 must be both a server and a client of the same RPC interface, when sending message to the peer, it is a client, when receiving message, the role is reversed, and it becomes the server.

What we are going to do demands more from both powerRPC and the programmer.

The talk RPC interface contains a single function: **send_msg()**.

```
interface talk2 {

  property   TRANSPORT_PROTOCOL = udp;
  property   GEN_MAIN_FUNC = false;
  property   SERV_CALL_PREFIX = s_;

  void  send_msg(
        unsigned long sender_program_no,
        char [size = strlen(sender_host) + 1, 1024] sender_host,
        char [size = strlen(sender_name) + 1, 1024] sender_name,
        char [size = strlen(msg) + 1, 4096] msg
  ) = 1;

} = 0x9999;
```

The **send_msg()** RPC takes four arguments, the first three identifies the sender, the last one is the message being sent. Since talk2 is both a server and a client, we must define the property **SERV_CALL_PREFIX**, so the server implementation of the **send_msg** is actually named **s_send_msg**. We also need to write our **main()** function, therefore the **GEN_MAIN_FUNC** property is set to false.

Implementation

Our code for s_send_msg() is just a little more than a few printfs.

```
void   s_send_msg(u_long sno, char *host, char *sender, char *msg)
{
    printf("\n..............%s@%d@%s.............\n%s",
           sender, sno, host, msg);
    printf("..................over.................\n");
    talk2_unbind(0);
    talk2_bind(host, sno, 0, 0);
}
```

Besides writing the message from the client onto the terminal, we also establish an RPC connection to the sender.

To receive messages from a peer, the Talk2 program needs to be an RPC server, to send messages it must read stdin and acts as an RPC client. This requires the Talk2 program to do I/O multiplexing, in our case, the Talk2 program must handle the input from both the network communication channels for RPCes and the terminal input. PowerRPC accommodates this easily by providing a set of server library functions to set up I/O handlers for a particular file descriptor.

Although we could use the **INIT_AFTER_REGISTER** property to insert all of the code, we take this chance to write the server main() function ourselves using the powerRPC generated code and libraries.

Thus we have the following code,

```
void    handle_stdin(int fd)
{
    char            msg[1024];
    int             cnt;
    cnt = read(fd, msg, 1023);
    if (cnt <= 0)
        exit(1);

    msg[cnt] = '\0';
    send_msg(myprog, myhost, myname, msg);
}


int     main(int argc, char **argv)
{
    fd_set          fds;
    SVCXPRT        *mytxp;
    int             serv_sock;
    int             dtsz;
    char            msg[4096];

    if (argc < 2) {
        printf("Usage: %s -n talker -p ", argv[0]);
        printf("program# [-c peer -P perr_prog#]\n");
        exit(0);
    }
    getoption(argc, argv);

    gethostname(myhost, 1023);

    if (strlen(peer_host)) {
        if (!talk2_bind(peer_host, peer_no, 0, 0)) {
            printf("Can not find peer to talk.\n");
            exit(1);
        }
    }
    pw_serv_init();
```

```
    if (!talk2_1_reg(RPC_ANYSOCK, myprog, 1, IPPROTO_UDP))
        exit(1);

    myprog = myprog == 0 ? TALK : myprog;

    printf("My program number is %d\n", myprog);
    signal(SIGINT, unset_myprog);

    pw_serv_input_handler(0, handle_stdin);

    pw_serv_mainloop(0, 0, 0, 0);
}
```

As you can the see, the handle_stdin() function simply reads something from stdin and call the send_msg() RPC to deliver it to a connected peer. The main server function is more interesting. First, it reads some options. To allow two talk2 programs to sit on the same machine, we let them to use whatever program number the user gives (if none is given at the command line option then the one defined in the IDL will be used). The first instance of Talk2 has no one to talk with, however, a subsequent Talk2 can talk to it by supplying the peer hostname and program number, through the ``-c'' and ``-P'' options respectively. When the **peer_host** variable is set, we try to make connection to another Talk2. Then we initialize the server code by calling pw_serv_init() function. Then we register our server by calling the generated talk2_1_reg() function, with the program number defined in myprog variable. We then set up the input handler for stdin. Finally, Talk2 enters its mainloop by calling pw_serv_mainloop() with default arguments.

Usage

The first Talk2 program would be started like this

```
% talk2 -n Mike -p 1234
```

It will announce ``My program number is 1234''.

Suppose the first one is on machine **host1**, we can talk with it by

```
%talk2 -n Dave -p 2345 -P 1234 -c host1
```

Now both Mike and Dave can type in messages on their terminal and talk to each other.

Our 100 line Talk2 is not intended to be a replacement for the existing talk program. But you can make it comparable in functionality with talk by writing more code. How good it can be is only limited by your C/C++ skills, not by your knowledge of networking or things like that.

Asynchronous RPC

Sometimes, we want an RPC to return immediately, and let the server return the results to the client asynchronously. For example, if it takes the quote server a long time to query a database to get the data, we should let the client to continue to do other things and get the result later when it is ready.

With powerRPC this can be achieved easily. By setting the NON_BLOCKING property to true, an RPC call would return before the server function gets called. To receive the result later, the client can register a collection service, which is an RPC interface. When the server get the result, it calls upon this RPC registered by the client to send the result back.

In this example, we demonstrate how to make our quote RPC asynchronous. We have two RPC interfaces defined in back.idl and over.idl. The client main function looks like this,

```
#include "over.h"
#include "back.h"
#include <signal.h>
#include <rpc/pmap_clnt.h>

main(int argc, char **argv)
{
    char            myhost[1024];
    if (argc < 3) {
        printf("usage: %s hostname ticker\n", argv[0]);
        exit(1);
    }
    pw_serv_init();

    back_1_unmap(0, 0);


    if (!back_1_reg(RPC_ANYSOCK, BACK, BACK_1, IPPROTO_TCP)) {
        printf("fail registering \n");
        exit(0);
    }
    pw_serv_async_enable();

    if (!over_bind(argv[1], 0, 0, 0)) {
        printf("fail connect to OVER server.\n");
        exit(0);
    }
    gethostname(myhost, 1023);
    getQuote(myhost, argv[2]);
    printf("Returned from 1st getQuote... now do something else ..\n");
    getQuote(myhost, argv[2]);
    printf("Returned from 2st getQuote... now do something else ..\n");
    getQuote(myhost, argv[2]);
    printf("Returned from 3rd getQuote... now do something else ..\n");

    while (1) {
        printf("happily doing ....\n");
```

```
        sleep(10);
    }
}
```

In the above, the client set up an asynchronous RPC back. Then it makes two calls of getQuote(), both return immediately. This getQuote() call is different from the one we studied earlier: it has an additional argument myhost. When the server receives the call, it gets the quote, and calls back myhost to send the results. The client enters a **while** loop, pretending to do other work, when the server's callback arrives, the client will be interrupted, and the returnQuote() implementation is called. The asynchronous behavior of the client's RPC is enabled by the pw_serv_async_enable() powerRPC library call .

Below is the server's code.

```
#include "back.h"
#include <stdlib.h>
/*
This is an asynchronous call. The client sends over its hostname to the
server, the server replies back to the client immediately.  Then client
and server reverse their roles, the server makes the RPC returnQuote()
to send the result to the client, which is supposed to acting as a BACK
server waiting for the result.
*/

void   getQuote(char *caller_host, char Ticker[8])
{

    stkQuote        quote;

    printf("Called by %s\n", caller_host);
    printf("sleep for a while ....\n");

    sleep(6);

    if (!back_bind(caller_host, 0, 0, 0)) {
        printf("client is not there!\n");
        return;
    }
    strcpy(quote.Ticker, Ticker);

    /* find the quotes */
    quote.Low = rand() % 100;
    quote.High = rand() % 100;
    quote.Close = rand() % 100;
    printf("now sending back quote ....\n");
    printf("%s %f %f %f\n", quote.Ticker,
           quote.Low, quote.High, quote.Close);

    returnQuote(&quote);

    back_unbind(0);
}
```

Other sample programs

The powerRPC distribution conatins other sample programs for demonstration purposes.

Chapter

6

Platform dependent issues

The powerRPC IDL compiler generates identical source code for all supported UNIX-like platforms. The generated code for WIN32 only differs slightly. The powerRPC runtime library encapsulates the platform dependent issues. PowerRPC is based on top of ONC RPC. On systems (such as SVR4, e.g., Solaris 2.5) which uses TIRPC (TLI networking code), you may need link to additional libraries such as libsocket and libnsl. On other systems which use the socket based ONC RPC, no additional libraries need to be linked.

On Windows NT/95/98/ME/2000/XP/2003/X64, one needs to start the portmapper program, or install the Pmapsvc service (NT/2K/XP/03/X64) only, which is supplied with the PowerRPC package.

Index