Introduction to programming RPC Tutorial

This page illustrates the basics of using remote procedure calls (RPC). We'll start with a remote procedure to add two numbers. This program will accept two numbers from the command line, parse the ascii text to convert them to numbers, then call a remote procedure to add them and print the results.

Step 1. Create the IDL

The first step involves defining the interface. This has to abide by Sun's format for its Interface Definition Language (IDL). An IDL is a file (suffixed with .x) which optionally begins with a bunch of type definitions and then defines the remote procedures. A set of remote procedures are grouped into a version. One or more versions are grouped into a program.

Traditionally, ONC RPC restricted us to remote procedures that accept a single parameter and return a single parameter. Later versions added an option to support multiple parameters, but we'll stick with the traditional mechanism to provide greatest compatibility. The lack of multiple parameters isn't really a problem since all we need to do is define a data structure that holds all the parameters we need.

In this example we have one type definition to define a structure that holds two integers: this will be our input parameter for the add function. Our interface will also have one version and one program. We have to assign a number to each function, version, and program. The function will be given an ID of 1. So will the version. The program number is a 32-bit number. Sun reserved the range from 0 to 0x1fffffff. We'll number this program 0x23451111.

The IDL, which we'll put in a file named add.x looks like:

```
struct intpair {
        int a;
        int b;
};

program ADD_PROG {
        version ADD_VERS {
            int ADD(intpair) = 1;
        } = 1;
} = 0x23451111;
```

We can compile this to test if we missed anything:

```
rpcgen -C add.x
```

The -C flag (upper-case C) tells rpcgen to generate C code that conforms to ANSI C. This is the default on some versions of rpcgen. If we look at the files that were generated, we'll see:

add.h

This is the header file that we'll include in both our client and server code. It defines the structure we defined (intpair) and typedefs it to a type of the same name. It also defines the symbols ADD_PROG (0x23451111, our program number) and ADD_VERS (1, our version number). Then it defines the client stub interface (add_1) and the interface for the server-side function that we'll have to write (add_1_svc). In the past (pre ANSI-C), the client stub and server side functions had the same name but since ANSI C was strict with parameters matching their declarations, this was changed since the parameters are slightly different. As we'll soon see, the client stub accepts an extra parameter representing a handle to the remote server. The server function gets an extra parameter containing information about who is making the connection.

add_svc.c

This is the server program. If you look at the code, you'll see that it implements the main procedure which registers the service and, if the symbol RPC_SVC_FG is not defined, forks a process to cause the service to run in the background; the parent exits.

The program also implements the listener for the program. This is the function named add_prog_1 (the _1 is used to distinguish the version number. The function contains a switch statment for all the remote procedures supported by this program and this version. In addition to the null procedure (which is always supported), the only entry in the switch statement is ADD, for our add function. This sets a function pointer (local) to server function, add_1_svc. Later in the procedure, the function is invoked with the unmarshaled parameter and the requestor's information.

add_clnt.c

This is client stub function that implements the add_1 function. It marshals the parameter, calls the remote procedure, and returns the result.

add_xdr.c

The _xdr.c file is not always generated; it depends on the parameters used for remote procedures. This file contains code to marshal parameters for teh intpair structure. It uses XDR (eXternal Data Representation) libraries to convert the two integers into a standard form.

Step 2. Generate sample client and server code

In addition to generating files to support remote procedure calls, rpcgen also has options to generate template code for both the client and server. This makes it easy to make sure that you create the RPC handle properly and call the procedures correctly. It even allows you to generate a makefile.

If we want rpcgen to generate everything, we can just run:

```
rpcgen -a -C add.x
```

This will create the client (add_client.c), the server (add_server.c), and the makefile (makefile.add or Makefile.add, depending on whether you are using SunOS or Linux; OS X will not create a makefile).

If you want to generate only the client template code, run:

```
rpcgen -Sc -C add.x >add_client.c
```

If you want to generate only the server function template code, run:

```
rpcgen -Ss -C add.x >add_server.c
```

If you want to generate only the makefile, run:

```
rpcgen -Sm -C add.x >makefile.add
```

Now we can compile our code by running:

```
make -f makefile.add (or Makefile.add)
```

To cut down on typing, let us rename makefile.add to makefile. This will allow us to simply type make with no parameters to recompile since by default make looks for a file named makefile or Makefile.

```
mv makefile.add makefile
```

If you need to change the name of the compiler to gcc because the default, cc, is not present on your system, you'll need to add a line to the makefile (for example, before the CFLAGS= line):

```
CC=gcc
```

Step 3. First test of the client and server

The template code written by rpcgen created a client program named add_client.c that accepts a single argument on the command line containing the name of the server, creates an RPC handle to the server process, and calls the add_1 function. A client template for an interface with more functions would contain a declaration of parameters and return values for each of the functions and call them one after the other.

The server function, contained in add_server.c is a function which does nothing but contains the comments:

```
/*
* insert server code here
*/
```

We will replace those comments with a single print statement:

```
printf("add function called\n");
```

Important!
Before we compile, we will make a change to the makefile. We will make sure that the server is compiled so that the symbol RPC_SVC_FG is defined. This will cause our server to run in the foreground. For testing purposes, this is convenient since we'll be less likely to forget about it and it will be easier to kill (we don't have to look up its process ID).

Edit the makefile and find the line that defines CFLAGS:

```
CFLAGS += -g
```

and change it to:

```
CFLAGS += -g -DRPC_SVC_FG
```

Secondly, we want to make sure that rpcgen generates code that conforms to ANSI C, so we'll add a -C (capital C) parameter to the rpcgen command. Change the line in the makefile that defines:

```
RPCGENFLAGS =
```

to:

```
RPCGENFLAGS = -C
```

Now compile your program by running make. You'll see output similar to:

```
cc -g -DRPC_SVC_FG -c -o add_clnt.o add_clnt.c
cc -g -DRPC_SVC_FG -c -o add_client.o add_client.c
cc -g -DRPC_SVC_FG -c -o add_xdr.o add_xdr.c
cc -g -DRPC_SVC_FG -o add_client add_clnt.o add_client.o add_xdr.o -lnsl
cc -g -DRPC_SVC_FG -c -o add_svc.o add_svc.c
cc -g -DRPC_SVC_FG -c -o add_server.o add_server.c
cc -g -DRPC_SVC_FG -o add_server add_svc.o add_server.o add_xdr.o -lnsl
```

Note that the -lnsl argument is not needed when linking under Linux, *BSD, or OS X.

If you're running OS X, Linux, or BSD and don't have a makefile, run the above commands using gcc as the compiler and omitting -lnsl:

```
gcc -g -DRPC_SVC_FG -c -o add_clnt.o add_clnt.c
gcc -g -DRPC_SVC_FG -c -o add_client.o add_client.c
gcc -g -DRPC_SVC_FG -c -o add_xdr.o add_xdr.c
gcc -g -DRPC_SVC_FG -o add_client add_clnt.o add_client.o add_xdr.o
gcc -g -DRPC_SVC_FG -c -o add_svc.o add_svc.c
gcc -g -DRPC_SVC_FG -c -o add_server.o add_server.c
gcc -g -DRPC_SVC_FG -o add_server add_svc.o add_server.o add_xdr.o
```

Unfortunately, the compilation produces a number of warnings but you can ignore them.
The result is that you have two executables: add_client and add_server. You can move add_server to another machine or run it locally, giving add_client your local machine's name or the name localhost.
If you're running this on a non-SunOS machine, you may need to start the RPC port mapper first. Chances are you won't and you should try running the server first. If it does not exit immediately with an unable to register error, you're probably good to go. If you do need to start the portmapper then here are the commands that you'll need to run on several popular operating systems:

```
/sbin/portmap
```

In one window, run:

```
sudo ./add_server
```

In another window, run:

```
./add_client localhost
```

In the server window, you should see the following text appear:

```
add function called
```

This confirms that the client and server are communicating. If your client just seems to be hanging, chances are you have not started the portmapper.

Step 4. Getting the server to do some work

Now that we know the client and sever compile and run, it's time to get the server to do some work. We will hard-code two numbers in the client that we want the server to add.

Initialize parameter in client

Edit add_client.c. Note that the function add_prog_1 defines a variable add_1_arg. This argument is passed as a parameter to the remote procedure call a few lines later with the call:

result_1 = add_1(&add_1_arg, clnt);

Before calling this function, we will initialize add_1_arg to contain the numbers 123 and 22. The variable is defined as type intpair, which we defined in add.x as containing to integers a and b (take a look at add.h to see how this is defined to the program). We can initialize the parameters with:

```
if (clnt == NULL) {
clnt_pcreateerror(host);
exit(1);
}
add_1_arg.a = 123;
add_1_arg.b = 22;
if (result_1 == NULL)  {
      clnt_perror(clnt, "call failed:");
 }
```

We can run make to make sure there were no syntax errors in transcribing these two lines (or the commands listed in step 3).

Turning to the server, add_server.c, we see that the first parameter of the add_1_svc function is our incoming parameter of type intpair.

We will add another print statement after the "add function called\n" one to print the values of the parameters. For more complex data types, these statements can serve as sanity checks and debugging aids. If the first statement is printed and the server dies after that, we know we made a mistake in dereferencing the parameters.
ONC RPC passes an address to the parameter on the client side and the server receives a local address of the incoming parameter. We now add the statement:

```
printf("add function called\n");
printf("parameters: %d, %d\n", argp->a, argp->b);
```

Compile and run this to see if we get what we expect. Compile with

make

and run the server (make sure you killed the old one) and then the client as in step 3. On the server window you should see:
parameters: 123, 22

This confirms that we get the parameters correctly. Let's compute the result and send it back. Before we return the address of result, set it to the sum of the two parameters and add another print statement:

```
printf("add function called\n");
printf("parameters: %d, %d\n", argp->a, argp->b);
result = argp->a + argp->b; printf("returning: %d\n", result);
return &result;
```

Note that the variable result is declared static. This is crucial because local (automatic) variables live on the stack. As soon as the server function returns the pointer to the result back to the server stub, the memory used by local variables can be reclaimed for use by the

server stub. Failure to declare the return type static can result in nasty bugs where the code may seem to work a lot of the time but not always.
On the client, we'll need to add code to print the result. The return value from the call to add_1 is a pointer to the result type (int in this case). This allows us to find out whether the remote procedure call succeeded or not. If the return value is a 0 (a null pointer), we know the RPC failed. Otherwise, we can dereference the return type and get the value. Let's modify add_client.c to print the value:

```
if (result_1 == (int *) NULL) {
clnt_perror (clnt, "call failed");
} else {
printf("result = %d\n", *result_1);
}
```

Compile (make) and run the code again on the server and client. As soon as you run the client code, you should see the following on the server:

```
add function called
parameters: 123, 22
returning: 145
```

And the following on the client:

```
result = 145
```

We now have a working server.

Step 5. Making the client functional

All we need now to get our program to work is to get the two numbers from the command line instead of using hard-coded values. In add_client.c we'll change the main function to accept three parameters: the server name, the first number, and the second number. The numbers will be parsed into integers and passed to add_prog_1.
Our main function used to look like:

```
main(int argc, char *argv[]){
        char *host;
         if (argc < 2)
        {
                printf("usage: %s server_host\n", argv[0]); exit(1);
        }
        host = argv[1];
        add_prog_1(host);
}
```

It now looks like:

```
int main(int argc, char *argv[])
        {
        char *host; int a, b; if (argc != 4)
                {
                printf ("usage: %s server_host num1 num2\n", argv[0]); exit(1);
                } host = argv[1];
                if ((a = atoi(argv[2])) == 0 && *argv[2] != '0')
                {
                fprintf(stderr, "invalid value: %s\n", argv[2]); exit(1);
                } if ((b = atoi(argv[3])) == 0 && *argv[3] != '0')
                {
```

```
                fprintf(stderr, "invalid value: %s\n", argv[3]); exit(1);
                }
                add_prog_1(host, a, b);
        }
```

We change add_prog_1 to accept two additional parameters:

```
void add_prog_1(char *host, int a, int b)
```

and set the parameters for the call to the remote procedure with:

```
add_1_arg.a = a;
add_1_arg.b = b;
result_1 = add_1(&add_1_arg, clnt);
```

Before compiling, add a `#include <stdio.h>` at the start of the file to define *stderr*, the file descriptor for the standard error output used by *stdio* functions (such as *printf*).

The complete client looks like:

```
#include "add.h"
#include <stdio.h>
void add_prog_1( char* host, int a, int b ){
        CLIENT *clnt; int *result_1;
        intpair add_1_arg; clnt = clnt_create(host, ADD_PROG, ADD_VERS, "udp");
        if (clnt == NULL) {
        clnt_pcreateerror(host); exit(1);
        }
        add_1_arg.a = a;
        add_1_arg.b = b;
        result_1 = add_1(&add_1_arg, clnt);
        if (result_1 == NULL) {
        clnt_perror(clnt, "call failed:");
        }
        else
                print("return is %d\n", *result_1);
                clnt_destroy( clnt );
                }

        int main(int argc, char *argv[])
        {
        char *host; int a, b; if (argc != 4) {
        printf ("usage: %s server_host num1 num2\n", argv[0]); exit(1);
        }
        host = argv[1];
        if ((a = atoi(argv[2])) == 0 && *argv[2] != '0') {
        fprintf(stderr, "invalid value: %s\n", argv[2]);
        exit(1);
        }
        if ((b = atoi(argv[3])) == 0 && *argv[3] != '0') {
        fprintf(stderr, "invalid value: %s\n", argv[3]);
        exit(1);
```

```
}
add_prog_1(host, a, b);
}
```

Compile (make) again, run the server, and run the client. For example:
./add_client localhost 5 7

should yield:

result = 12
The program now fully works!

Step 6. Cleanup

Your program is now working. All it needs now is some cleaning up. This is a crucial step for any non-throw-away code. You want to make sure the program is readable and flows well. It shouldn't have the appearance that you just hacked some code from a template just to get it working. The overall style should be consistent between your code and the rpcgen-generated code. For example:

- I do not add a space before the opening parenthesis on a function call (e.g. "clnt_destroy(clnt)" instead of "clnt_destroy (clnt)", so I removed them here.
- A name such as `add_1_arg` is not only unclear but rather long. It's as bad as naming the first parameter to function `f` as `f_arg`. I'm just going to call it `v` (for value; the code is so short that the meaning is clear and doesn't need the verbosity that a more global variable would call for).
- It's important that you exit the program or otherwise ensure that you *do not make any more RPC calls* if you received an error from an RPC call (a return of 0).
- I like having my *main* function at the top, so I moved it up and declared the function prototypes above it.
- I moved the call to *clnt_destroy* to the *main* function. This allows one to call *add* multiple times without having it destroy the client handle after the first call.
- I created a function called *rpc_setup* to create the client handle. It's only a few calls but it makes the main function cleaner to follow.
- I added `#include <stdio.h>` at the top of the file to avoid errors of an undefined *stdio.h* (some systems gripe more than others).
- I added `#include <stdlib.h>` at the top of the file to keep the compiler on my mac from giving warnings about the implicit definition of *exit*. You may not need this on other systems.
- I removed any comments about this being template code.

The #ifdef DEBUG statements should be removed and the code can be made easier to read if the *main* function is moved to the top. Variables can be renamed more sensibly (add_1_arg is a bit crude). Auto-generated comments should be removed.

If we really intended to use the add remote procedure call repeatedly we would not want to create the handle each time, so it would be a good idea to create the RPC handle just once.

Here is the final client code:

```
/* RPC example: add two numbers */

#include "add.h"

CLIENT *rpc_setup(char *host);
```

```
void add(CLIENT *clnt, int a, int b);

int main(int argc, char *argv[])
{
        CLIENT *clnt;  /* client handle to server */
        char *host;    /* host */
        int a, b;

        if (argc != 4) {
                printf("usage: %s server_host num1 num2\n", argv[0]);
                exit(1);
        }
        host = argv[1];
        if ((a = atoi(argv[2])) == 0 && *argv[2] != '0') {
                fprintf(stderr, "invalid value: %s\n", argv[2]);
                exit(1);
        }
        if ((b = atoi(argv[3])) == 0 && *argv[3] != '0') {
                fprintf(stderr, "invalid value: %s\n", argv[3]);
                exit(1);
        }
        if ((clnt = rpc_setup(host)) == 0)
                exit(1);       /* cannot connect */
        add(clnt, a, b);
        clnt_destroy(clnt);
        exit(0);
}

CLIENT *rpc_setup(char *host)
{
        CLIENT *clnt = clnt_create(host, ADD_PROG, ADD_VERS, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror(host);
                return 0;
        }
        return clnt;
}

Void add(CLIENT *clnt, int a, int b)
{
        int  *result;
        intpair v;     /* parameter for add */

        v.a = a;
        v.b = b;
        result = add_1(&v, clnt);
        if (result == 0) {
                clnt_perror(clnt, "call failed");
        } else {
                printf("%d\n", *result);
        }
}
```

And the server (with one line of diagnostics being printed) is:

```
/* * RPC server code for the remote add function */

        #include "add.h"

        int * add_1_svc(intpair *argp, struct svc_req *rqstp) {
```

```
        static int result;

        result = argp->a + argp->b;

        printf("add(%d, %d) = %d\n", argp->a, argp->b, result);

        return &result;

        }
```

Even for a trivially simple program like this there are several ways to format the output. I opt for a single number on one line:

```
        167
```

the number on a single line rather than something like:

```
        result is 167
```

or

```
        RPC program 123+44 = 167
```

Question: Why?

One last thing

Before we're completely done with this, let's create a new makefile. Unfortunately the automatically-generated one is neither easy to read nor reliable in its list of dependencies. We'll write a very explicit one. This is how our makefile looks now:

```
CC = gcc
CFLAGS = -g -DRPC_SVC_FG
RPCGEN_FLAG = -C

all: add_client add_server

# the executables: add_client and add_server

add_client: add_client.o add_clnt.o add_xdr.o
        $(CC) -o add_client add_client.o add_clnt.o add_xdr.o –lnsl

 add_server: add_server.o add_svc.o add_xdr.o
        $(CC) -o add_server add_server.o add_svc.o add_xdr.o -lnsl

# object files for the executables

add_server.o: add_server.c add.h
        $(CC) $(CFLAGS) -c add_server.c
add_client.o: add_client.c add.h
        $(CC) $(CFLAGS) -c add_client.c
```

```
# compile files generated by rpcgen
add_svc.o: add_svc.c add.h
	$(CC) $(CFLAGS) -c add_svc.c add_clnt.o: add_clnt.c add.h
	$(CC) $(CFLAGS) -c add_clnt.c add_xdr.o: add_xdr.c add.h $(CC) $(CFLAGS) -c add_xdr.c

# add.x produces add.h, add_clnt.c, add_svc.c, and add_xdr.c

# make sure we regenerate them if our interface (add.x) changes
add_clnt.c add_svc.c add_xdr.c add.h: add.x
	rpcgen $(RPCGEN_FLAG) add.x

clean:
	rm -f add_client add_client.o add_server add_server.o add_clnt.* add_svc.* add.h
```