# Remote Procedure Call

**( csc6320 survey paper )**

**Yong Li**

**Dec. 02, 2000**

## ABSTRACT

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client/server based applications. This survey is to explain RPC mechanism, how to use the Sun RPC convert local procedures to remote procedures.


## 1. Introduction to Client/Server Distributed Architecture

Many modern data processing applications utilize “distributed computing architecture” in which a user must access data or service across a network. Client/server is a computational architecture that involves client processes requesting service from server processes. The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s. The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve, usability, flexibility, interoperability, and salability as compared to centralized, mainframe, time sharing computing.

The logical extension of this is to have clients and servers running on the appropriate hardware and software platforms for their functions. For example, database management system servers running on platforms specially designed and configured to perform queries, or file servers running on platforms with special elements for managing files.

The client is a process (program) that sends a message to a server process (program), requesting that the server perform a task (service). Client programs usually manage the user-interface portion of the application, validate data entered by the user, dispatch requests to server programs, and sometimes execute business logic. The client-based process is the front- end of the application that the user sees and interacts with. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The

client process also manages the local resources that the user interacts with such as the monitor, keyboard, workstation CPU and peripherals. One of the key elements of a client workstation is the graphical user interface (GUI). Normally a part of operating system i.e. the window manager detects user actions, manages the windows on the display and displays the data in the windows.

A server process (program) fulfills the client request by performing the task requested. Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity and dispatch responses to client requests. Sometimes server programs execute common or complex business logic. The server-based process "may" run on another machine on the network. This server could be the host operating system or network file server; the server is then provided both file system services and application services. Or in some cases, another desktop machine provides the application services. The server process acts as a software engine that manages shared resources such as databases, printers, communication links, or high powered-processors. The server process performs the back-end tasks that are common to similar applications.

A client is defined as a requester of services and a server is defined as the provider of services. The client (user system) sends a message to a server requesting the server to perform a task or service. The client programs perform tasks in the user interface portion of the application. They validate data entered by the user and send the request to the server programs.

The basic characteristics of client/server architectures are:

These are the common characteristics of client/server architectures

- A client who interacts with the user and a server, which interacts with the resources.
- The computing resources needed by the client and the server like memory, processor speed, disk speeds etc. are different.
- The hardware platform and operating system of client and server can be heterogeneous
- An important characteristic of client-server systems is salability.

What is distributed function processing?

The distribution of applications and business logic across multiple processing platforms. Distributed processing implies that processing will occur on more than one processor in order for a transaction to be completed. In other words, processing is distributed across two or more machines and the processes are most likely not running at the same time, i.e. each process performs part of an application in a sequence. Often the data used in a distributed processing environment is also distributed across platforms. Here the split occurs in the application functionality, one part going to the client, other to the server. The underlying communications facilities may implement either a message-based or remote procedure call (RPC) mechanism for transfer of dialog and data.

## 2. RPC Overview

The Remote Procedure Calls (RPC) mechanism is a high-level communications paradigm for network applications. RPC is a powerful technique for constructing distributed, client/server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

By using RPC, programs on networked platforms can communicate with remote (and local) resources. RPC allows network applications to use specialized kinds of procedure calls designed to hide the details of underlying networking mechanisms. The complexity involved in the development of distributed processing is reduced by keeping the semantics of a remote call the same whether or not the client and server are collocated on the same system.

The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

## 3. How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

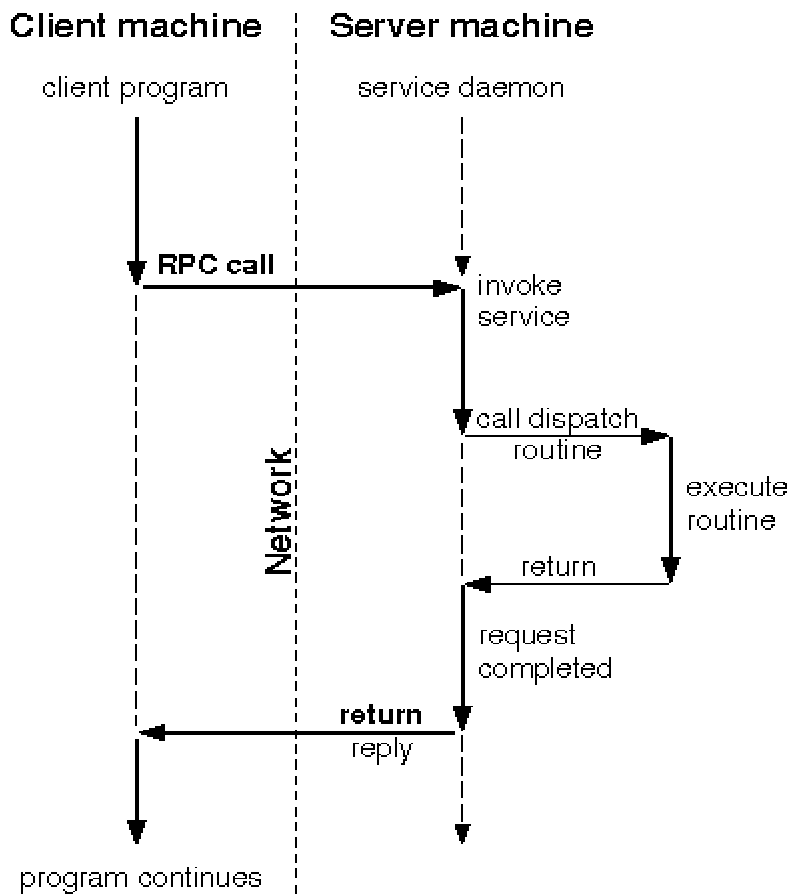


**Figure 1. Remote Procedure Calling Mechanism**

## 4. RPC Application Development

To develop an RPC application the following steps are needed:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program

The programs will be compiled separately. The communication protocol is achieved by generated stubs and these stubs and rpc (and other libraries) will need to be linked in.

### 4.1 The rpcgen Protocol Compiler

The easiest way to define and generate the protocol is to use a protocol compiler such as rpcgen. rpcgen provides programmers a simple and direct way to write distributed applications. rpcgen does most of the dirty work, allowing programmers to debug the main features of their application, instead of requiring them to spend most of their time debugging their network interface code.

rpcgen is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output which includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions. The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are to be invoked by remote clients. rpcgen's output files can be compiled and linked in the usual way. The developer writes server procedures -- in any language that observes Sun calling conventions -- and links them with the server skeleton produced by rpcgen to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by rpcgen. Linking this program with rpcgen's stubs creates an executable program. (At present the main program must be written in C). rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server stub.

The output of rpcgen is:

- A header file of definitions common to the server and the client
- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server

- A stub program for the client
- A time-out for servers (optionally)
- C-style arguments passing ANSI C-compliant code  (optionally)
- (optionally) dispatch tables that the server can use to check authorizations and then invoke service routines.

### 4.2 RPC specification

A file with a ``.x'' suffix acts as a remote procedure specification file. It defines functions that will be remotely executed functions. Functions are restricted: they may take at most one in parameter, and return at most one out parameter as the function result.
If you want to use more than one in parameter, you have to wrap them up in a single structure, and similarly with the out values.
Multiple functions may be defined at once. They are numbered from one upward, and any of these may be remotely executed.
The specification defines a program that will run remotely, made up of the functions. The program has a name, a version number and a unique identifying number (chosen by you).

RPC Language Specification likes this:

```
program-def:
         "program" identifier "{"
             version-def
             version-def *
         "}" "=" constant ";"

     version-def:
         "version" identifier "{"
             procedure-def
              procedure-def *
          "}" "=" constant ";"

     procedure-def:
```

```
type-specifier identifier "(" type-specifier ")"
"=" constant ";"
```

### 4.3 RPC versions and numbers

Each RPC procedure is uniquely identified by a program number, version number, and procedure number.

The program number identifies a group of related remote procedures, each of which has a different procedure number. Program numbers are given out in groups of hexadecimal 20000000.

```
0 - 1fffffff defined by Sun
20000000 - 3fffffff defined by user
40000000 - 5fffffff transient
60000000 - 7fffffff reserved
80000000 - 9fffffff reserved
a0000000 - bfffffff reserved
c0000000 - dfffffff reserved
e0000000 - ffffffff reserved
```

Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number does not have to be assigned. The first implementation of a program will most likely have version number 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies the version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible. Version numbers are incremented when functionality is changed in the remote program. Existing procedures can be changed or new ones can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification.

### 4.4 Stubs

When the calling process calls a procedure, the action performed by that procedure will not be the actual code as written, but code that begins network communication. It has to connect to the remote machine, send all the parameters down to it, wait for replies, do the right thing to the stack and return. This is the client side *stub*.

The server side stub has to wait for messages asking for a procedure to run. It has to read the parameters, and present them in a suitable form to execute the procedure locally. After execution, it has to send the results back to the calling process.

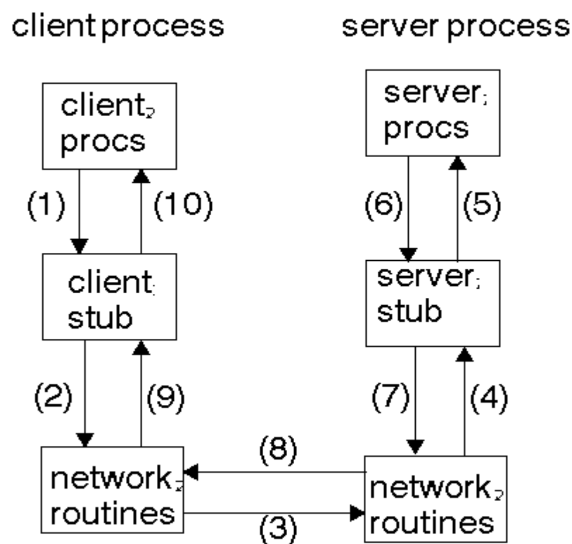


**Figure 2. Stubs**

1. The client calls the local stub procedure. The stub packages up the parameters into a network message. This is called *marshaling*.
2. Networking functions in the O/S kernel are called by the stub to send the message.

3. The kernel sends the message(s) to the remote system. This may be connection-oriented or connectionless.
4. A server stub unmarshals the arguments from the network message.
5. The server stub executes a local procedure call.
6. The procedure completes, returning execution to the server stub.
7. The server stub marshals the return values into a network message.
8. The return messages are sent back.
9. The client stub reads the messages using the network functions.
10. The message is unmarshalled. And the return values are set on the stack for the local process.

### 4.5 Data representation

A procedure, for example, may have a short int, a string and an ordinary int as parameters. How is it to be marshaled so that it can be correctly unmarshaled at the other end?

For example, the short int could use the first two bytes with the next two blanks, or the other way round. The string could be prefixed by its length or be terminated by a sentinel value. If the length is sent, should it be an int? A short int? The ordinary int could be big-endian or little-endian.

The Sun RPC uses a standard format called **XDR**(e**X**ternal **D**ata **R**epresentation), a protocol for the machine-independent description and encoding of data. . XDR is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type. XDR is useful for transferring data between different computer architectures.

RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to XDR representation before sending

them over the wire. The process of converting from a particular machine representation to XDR format is called ***serializing***, and the reverse process is called ***deserializing.***

Valid data types supported by XDR include:

- `int`
- `unsigned int`
- `long`
- `enum`
- `bool`
- `float`
- `double`
- `typedef`
- `structure`
- `fixed array`
- `string (null terminated char *)`

## 5. Converting Local Procedures into Remote Procedures

Assume an application that runs on a single machine. Suppose we want to convert it to run over the network. Here we will show such a conversion by way of a simple example program that calculates pi.

To declare a remote procedure, first, determine the data types of all procedure-calling arguments and the result argument. The calling argument of calcu_pi () is a void, and the result is a double. We can write a protocol specification in RPC language that describes the remote version of calcu_pi. The RPC language source code for such a specification is:

```
/* pi.x: Remote pi calculation protocol */

program PIPROG {
    version CALCU_PIVERS {
        double CALCU_PI() = 1;
    } = 1;
} = 0x39876543;
```

Remote procedures are always declared as part of remote programs. The code above declares an entire remote program that contains the single procedure CALCU_PI.

In this example,

- CALCU_PI procedure is declared to be:
    - the procedure 1,
    - in version 1 of the remote program
- PIPROG, with the program number `0x39876543`.

Notice that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow.

To compile a .x file using

```
rpcgen –a –C pi.x
```

where: option –a tells rpcgen to generate all of the supporting files
option –C indicates ANSI C is used

This will generate the files:

- pi_clnt.c -- the client stub
- pi_svc.c -- the server stub
- pi.h -- the header file that contains all of the XDR types generated from the specification
- makefile.pi -- makefile for compiling all of the client and server code
- pi_client.c -- client skeleton, need to be modified
- pi_server.c – server skeleton, need to be modified

Sun RPC supports the passing of only a single parameter and the return of only a single result. If more than one argument is required, the arguments must be passed in a struct. rpcgen may also be used to generate XDR routines, i.e., the routines necessary to convert local data structures into XDR format and vice-versa.

The following example a remote adding and subtracting two integers service,  built using rpcgen not only to generate stub routines, but also to generate the XDR routines.

```
/** simp.x
 *  the definition of the data type that will be passed to
 *  both of the remote procedures add() and sub()
 */

struct operands {
      int x;
      int y;
};

/* program, version and procedure definitions.
 */

program SIMP_PROG {
   version SIMP_VERSION {
     int ADD(operands) = 1;
     int SUB(operands) = 2;
   } = 1;
} = 0x28976543;
```

Running **rpcgen** on *simp.x* not only creates the header file, client stub routines and server skeleton. It also creates the XDR routines necessary for converting instances of declared data types from host platform representation into XDR format, and vice-versa. These routines are output in the file *simp_xdr.c*.

Now, we come back the first example.  There are just two more files to modify:

- the remote procedure itself, pi_server.c
- the main client program that calls it, pi_client.c

Here's one possible definition of a remote procedure to implement the CALCU_PI procedure we declared previously:

```
/*
 * pi_server.c: implementation of the remote procedure "calcu_pi"
 *
 * The formula is:
 * (pi / 4) = 1 - 1/3 + 1/5 - 1/7 ...
 */
```

```
#include <rpc/rpc.h> /* always needed */
#include "pi.h"      /* generated by rpcgen compiler */

/** Remote verson of "calcu_pi" */
double * calcu_pi_1_svc(void *argp, struct svc_req *rqstp){
    static double  pi;

    double sum = 0;
    int i;
    int sign;

    for (i=1; i<10000000; i++ ){
        sign = (i+1) % 2;
        if ( sign == 0 )
            sign = 1;
        else
            sign = -1;
        sum += 1 / (2*(double)i -1) * (double)sign;
    }
    pi = 4 * sum;
    return (&pi);
}
```

Notice that the declaration of the remote procedure calcu_pi_1() differs from that of the local procedure in several ways:

- It always takes pointers to their arguments instead of arguments themselves. This is true of all remote procedures.
- It returns a pointer to a double instead of a double itself. This is also generally true of remote procedures: they always return a pointer to their results.
- It has an `_1' appended to its name. In general, all remote procedures called by `rpcgen' are named by the following rule: the name in the program definition (here `CALCU_PI') is converted to all lower-case letters, an underbar (`_') is appended to it, and finally the version number (here `1') is appended.
- When **rpcgen** is used, it is essential to have result (in this example) declared as static. In the code generated by **rpcgen**, the result address is converted to XDR format *after* the remote procedure returns. If the result were declared local to the remote procedure, references to its address would be invalid after the remote procedure returned. So the result *must* be declared static when **rpcgen** is used.

The last thing to do is declare the main client program that will call the remote procedure. Here is one possibility:

```
/** pi_client.c, remote version of client program
 */

#include <stdio.h>
#include <rpc/rpc.h>   /* always needed */
#include "pi.h"        /* generated by rpcgen*/

main(int argc, char *argv[])
{
    CLIENT *clnt;
    double  *result_1;
    char *host;
    char *  calcu_pi_1_arg;

    /* must have two arguments, including server host name */
    if (argc < 2) {
      printf("usage:  %s server_host\n", argv[0]);
      exit(1);
    }

    host = argv[1]; /* server host name */

    /** Create client "handle" used for calling PIPROG
     *  on the server designated on the command line.
     */
    clnt = clnt_create(host, PIPROG, CALCU_PIVERS, "tcp");

    if (clnt == (CLIENT *) NULL) {
      /* Couldn't establish connection with server
      *  Print error message and die.
      */
      clnt_pcreateerror(host);
      exit(1);
    }

    /* call remote procedure on the server side */
    result_1 = calcu_pi_1((void *)&calcu_pi_1_arg, clnt);

    if (result_1 == (double *) NULL) {
      /** An error occurred while calling the server.
      *  Print error message and die.
      */
      clnt_perror(clnt, "call failed");
      exit(1);
    }

    /** Okay, we successfully called the remote procedure.
    * print the pi value
    */
```

```
    printf("PI is %f\n" , *result_1);
    clnt_destroy(clnt);
    exit(0);
}
```

There are four things to note here:

- First a client *handle* is created using the RPC library routine clnt_create(). This client handle will be passed to the stub routines that call the remote procedure.
- The last parameter to clnt_create() is ``tcp''.
- The remote procedure calcu_pi_1() is called exactly the same way as it is a local procedure except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result itself.
- The remote procedure call can fail in two ways. The RPC mechanism itself can fail or, alternatively, there can be an error in the execution of the remote procedure. In the former case, the remote procedure returns with a NULL. In the later case, however, the details of error reporting are application dependent. Here, the error is being reported via *result.

Here's how to compile the programs:

```
$ rpcgen –C –a pi.x
$ gcc pi_client.c pi_clnt.c -o pi_client -lnsl
$ gcc pi_server.c pi_svc.c -o pi_server –lnsl
```

Two programs are compiled here: the client program **pi_client** and the server program **pi_server**. Once created, the server should be copied to a remote machine and run. (If the machines are homogeneous, the server can be copied as a binary. Otherwise, the source files will need to be copied to and compiled on the remote machine.) For this example, the remote machine is called *remote* and the local machine is called *local*. The server is started from the shell on the remote system:

```
remote$ pi_server
```

Thereafter, a user on *local* can calculate pi as follows:

```
local$ pi_client  remote
```

In summary, follow these steps in converting local calls to remote calls:

- Get the program to work using local functions.
- Restructure each functions so it has only one parameter which is passed by values, and be sure that it works when called locally,
- Create a specification file having a .x extension.
- Call repgen with the –a and –C options to generate a complete set of files.
- Modify the calling program generated by rpcgen ( _client.c ).
- Modify the server program generated by rpcgen( _server.c ).
- Compile the modified filed linking with the stub files.
- Put server file on the remote machine and run.
- Run client file on local machine.

## 6. Conclusion

Sun rpcgen is a very powerful compiler. It can provide programmers a simple and direct way to write distributed applications. rpcgen' reduces development time that would otherwise be spent coding and debugging low-level routines. Since Sun RPC supports the passing of only a single parameter and the return of only a single result. If more than one argument is required, the arguments must be passed in a struct. Programmers should pay more attention on this restriction.

APPEDIX

```
/** simp.x
 *  the definition of the data type that will be passed to
 *  both of the remote procedures add() and sub()
 */

struct operands {
      int x;
      int y;
};

/* program, version and procedure definitions.
 */

program SIMP_PROG {
   version SIMP_VERSION {
     int ADD(operands) = 1;
     int SUB(operands) = 2;
   } = 1;
} = 0x28976543;
```

```
/* simp_server.c, definition of the remote add and subtract procedure used by
 * simple RPC example
 * rpcgen will create a template for you that contains much of the code
 * needed in this file is you give it the "-Ss" command line arg.
*/

#include "simp.h"

/* Here is the actual remote procedure */
/* The return value of this procedure must be a pointer to int! */
/* we declare the variable result as static so we can return a
   pointer to it */

int *
add_1_svc(operands *argp, struct svc_req *rqstp)
{
        static int  result;
        printf("Got request: adding %d, %d\n", argp->x, argp->y);
        result = argp->x + argp->y;
        return (&result);
}



int *
sub_1_svc(operands *argp, struct svc_req *rqstp)
{
        static int  result;
```

```
        printf("Got request: subtracting %d, %d\n", argp->x, argp->y);
        result = argp->x - argp->y;
        return (&result);
}
```

```
/* simp_client.c RPC client for simple addition and subtraction example */

#include <stdio.h>
#include "simp.h"  /* Created for us by rpcgen - has everything we need ! */

/* Wrapper function takes care of calling the RPC procedure */
int add( CLIENT *clnt, int x, int y) {
  operands ops;
  int *result;

  /* Gather everything into a single data structure to send to the server */
  ops.x = x;
  ops.y = y;

  /* Call the client stub created by rpcgen */
  result = add_1(&ops,clnt);
  if (result==NULL) {
    fprintf(stderr,"Trouble calling remote procedure\n");
    exit(0);
  }
  return(*result);
}

/* Wrapper function takes care of calling the RPC procedure */
int sub( CLIENT *clnt, int x, int y) {
  operands ops;
  int *result;

  /* Gather everything into a single data structure to send to the server */
  ops.x = x;
  ops.y = y;

  /* Call the client stub created by rpcgen */
  result = sub_1(&ops,clnt);
  if (result==NULL) {
    fprintf(stderr,"Trouble calling remote procedure\n");
    exit(0);
  }
  return(*result);
}


int main( int argc, char *argv[]) {
  CLIENT *clnt;
  int x,y;
```

```
  if (argc!=4) {
    fprintf(stderr,"Usage: %s hostname num1 num\n",argv[0]);
    exit(0);
  }

  /* Create a CLIENT data structure that reference the RPC
     procedure SIMP_PROG, version SIMP_VERSION running on the
     host specified by the 1st command line arg. */

  clnt = clnt_create(argv[1], SIMP_PROG, SIMP_VERSION, "udp");

  /* Make sure the create worked */
  if (clnt == (CLIENT *) NULL) {
    clnt_pcreateerror(argv[1]);
    exit(1);
  }

  /* get the 2 numbers that should be added */
  x = atoi(argv[2]);
  y = atoi(argv[3]);


  printf("%d + %d = %d\n",x,y, add(clnt,x,y));
  printf("%d - %d = %d\n",x,y, sub(clnt,x,y));
  return(0);
}
```