Remote Procedure Calls:

A Programming Overview Recitation for ECES 338 March 29, 1999

RPC programming is somewhat difficult to learn because it attempts to implement a very abstract mechanism for calling functions over a network. In essence, a typical RPC project breaks down into three main areas:

1. Using the `rpcgen` compiler, in conjunction with a protocol definition file, in order to generate the client and server backbone of the program.
2. Creating the code to be executed by the client(s).
3. Creating the code to be executed by the server.

In this lecture, I will attempt to highlight each of these areas, without going into specific coding examples (this will be presented later, after the actual assignment has been released).

RPCGEN and Protocol Definition Files:

The first step in that must be completed in order to use Remote Procedure Calls is to program a "protocol definition" specification. Using the specification, a special utility called `rpcgen` can be used in order to simplify (or confuse) the RPC process. The `rpcgen` utility will create several files, depending upon the options that you give it. The method for using this program is fully documented on pages 229 – 235 of *The Monkey Book*. I will only hit the highlights here.

First, I will attempt to show some of the capabilities of the *protocol definition* file. The purpose of this file is to lay the groundwork for your program. Basically, you will identify which functions that you want the RPC clients to be able to access from the RPC servers. This file, as given to `rpcgen`, uses a special syntax, which is a combination of C and Pascal syntax. Appendix C (page 441) of *The Monkey Book* contains a complete description of this language. In order to examine this language in some detail, let's concentrate on the following example:

```
const ERROR = -1;                                       /* 1 */

program SOME_PROGRAM_NAME {                             /* 2 */
      version SOME_VERSION_LITERAL {                    /* 3 */
            int some_function ( some_parameter ) = 1;   /* 4 */
      } = 1;                                            /* 5 */
} = 0x2000000x;                                         /* 6 */
```

Line one of the example creates a literal constant. Basically, `rpcgen` will simply encode this into the resulting header file as a `#define`, so that is how you should think of the `const` directive. Line two starts the actual meat of this file, with the program directive. All that you need to do is specify some literal that follows logically from the content of your program. The identifier that you specify will be used later, so that your client can attach to the proper RPC server. As shown in line six, this identifier stand for a number, which follows the structure definition. This number is documented in detail on page 231 of *The Monkey Book*, but in general it must start in the range shown, and be as unique as possible.

The `version` directive, as specified in line three, specifies a literal that is representative of the version of the code that is being defined. The purpose of this directive is to allow multiple versions of the same program to exist in a shared environment. The number (and literal) should be changed whenever you want your clients to be able to discern between different servers. The literal that you specify will be used again in your client program. Furthermore, as shown in line five, an actual number will back up this literal. This number will be integrated into the names of the client and server stub functions that are used by RPC in order to transfer data.

The final part of this example to be scrutinized is the actual function definition, as demonstrated in line four. In essence, a series of functions can be prototyped here. Each function can have either one or no (`void`) parameters. Also, it is typical for any functions defined to have an integer return type, so that the caller can detect a failure condition. Note that multiple parameters can be sent by encoding them into a buffer. However, this method is tricky, and requires some fancy maneuvers with the protocol definition file. See the example program (*producer/consumer*) that I posted to the web for more detail.

Once this file has been created, it can be fed to `rpcgen` in order to create all of the back-end files necessary in order to use RPC. This program takes a whole slew of parameters, but I want to make note of only two of them here. The first parameter is "`-C`", which will generate ANSI C code as a result of processing. This parameter is pretty useful, as your program probably won't work without it. The second class of parameters tells `rpcgen` to create example files, to aid you in the coding process. Here is a breakdown of these parameters:

`-Sc`	Generate sample client code that uses remote procedure calls.
`-Sm`	Generate a sample *Makefile* that can be used for compiling the application.
`-Ss`	Generate sample server code that uses remote procedure calls.

You may wish to take advantage of these parameters, in order to ease your programming experience. They can provide a helpful starting point, as you begin to enter the world of RPC programming.

Coding a Client:

A *client*, in the RPC world should be a separate program that connects to an RPC *server*. Basically, the client program will be able to perform its activities independently from any other client programs. Each client will use server-provided functions in order to access and/or modify shared data. In order to create a client program that can attach to an RPC server, three different functions are needed:

- `CLIENT *clnt_create (const char *host, const u_long prognum, const u_long versnum, const char *nettype)`
  - This is the *big* function necessary in order to enable a client program to connect to an RPC server. Based upon the given parameters, this function will return a pointer to a CLIENT structure, which represents a client *handle*. This handle will be used with other RPC functions, as well as with the remote functions that are controlled by the server. As such, the first parameter is a string that should contain the *hostname* of the server with which to connect. It is up to the client program to obtain this data, preferably as a command-line parameter. The next two parameters should be the identifiers for `program` and `version`, respectively, that you defined in your protocol definition file. The last parameter specifies the type of network service to be used for the RPC connection. A brief description of this parameter is given on page 237 of *The Monkey Book*.

- `void clnt_pcreateerror (const char *s)`
  - This function prints a message to standard error indicating why a client RPC handle could not be created. It should be used immediately after a call to `clnt_create()`, to report on a connection failure.

Once the RPC connection has been established, two more functions may be of use in order to complete the client program:

- `void clnt_destroy (CLIENT *clnt)`
  - This function removes a given client handle from the system, after the client program decides that it is done with it.

- `void clnt_perror(const CLIENT *clnt, const char *s)`
  - This is basically the RPC version of `perror()`. Be sure to give it a client handle, in order to ensure proper operation.

Coding a Server:

In order to implement an RPC server, no special functions are required. Basically, all of the functions specified in the protocol description file need to be fully defined. It should be noted, however, that the names of the functions would be significantly mangled by `rpcgen`. For example, using the example above, the function specified as `some_function`, would have to coded under its new name, `some_function_1_svc`. Finally, it should also be noted that since this is a concurrent environment, some sort of synchronization method (such as semaphores) would probably be necessary in order to ensure the integrity of the data.