

CSIS0234B Computer and Communication Networks (Class B)

Reading for Tutorial 4

Introducing RPC (Remote Procedure Calls)

Sockets allow programs to communicate through the network. It provides a low-level interface to exchange bytes. However, application programs communicate application layer messages, which usually contains larger, variable sized data structures made up of multiple bytes. Programmers usually find themselves having to differentiate various kinds of messages, determine how variable length strings or arrays should be stored, convert between big-endian and little-endian, etc.

As usual, no programmer want to spend time for boring repetitive tasks, so tools are built so that the problem can be solved once and for all. Using these tools, the programmers write descriptions about what services servers provide, what arguments each service requires, what are the type of each argument, etc. The tools then use standardized ways to perform the network communication. Programmers typically do not need to know how the communication is achieved. We will look at one particularly simple tool called RPC (Remote Procedure Call). It is the basis on which the Portmapper and Network Filesystem (NFS) are built upon.

1. The abstraction of RPC

Every programmer knows what happen when a function is called: the caller prepares arguments, calls the function, waits until the function finishes, which returns some results to the caller. This model is adopted by RPC. In particular, when a program wants to get service from a server, it prepares some arguments and makes a **remote** procedure call (RPC). Then it waits until the RPC finishes, which returns some results to the caller. As in normal function calls, many types of data can be passed between the caller and the callee. They are converted to a standardized byte-oriented representation (called External Data Representation, or **XDR**) by the RPC library. This conversion process is usually called **marshaling**.

To use RPC, the programmer writes a specification about what services a server provides, in a language called the “RPC Language” (RFC 1831, RFC1014). A standard program `rpcgen` is then executed on the specification, generating some “**stub functions**”, functions that perform marshaling and network communication. At least three files are generated, one is a common header file used by both the server and the client, the remaining two contains the stub functions for the server and the client respectively. The server stub file also contains a `main()` function, which binds itself a ephemeral port waiting for clients to connect to it, and inform the **portmapper** running at TCP/UDP port 111 about the port that is being used for the service so that clients can find the correct port. (As noted above, portmapper itself is implement with RPC.)

Then the programmer writes functions which actually **implements the declared services**, and link it with the server stub file to form a full executable server. He then writes client programs which use the client stub functions to obtain network service, and link it to the client stub file to produce an executable client.

2. Developing an RPC application

Let’s show the complete process to develop an extremely simple “hello-world” application using RPC. To do so we first write a specification `hello.x`:

```
program HELLOWORLD {
    version HELLOWORLDVERS {
        int PRINTMSG(string) = 1;
    } = 1;
} = 0x20000001;
```

Here HELLOWORLD and HELLOWORLDVERS are symbols we define, to identify the program and its version, with the value specified to be 0x20000001 and 1 respectively (the client gives these values to the portmapper, which is then used to locate the correct server port). The value of the program/version pair must be unique among all RPC programs running in the same system. For user applications, the numbers between 0x20000000 to 0x3fffffff may be used. (Numbers from 0 to 0x1fffffff are administered by Sun Microsystems, the initial developer of RPC. Numbers larger than 0x40000000 are reserved for future use.) Within the version specification is a list of services provided. Here only the service numbered 1 (PRINTMSG) is provided, taking a C-style string (not a C++ string!) as argument, returning an integer. If desired, more complicated types like structs can be defined and used. Recent versions of RPC (when rpcgen is used with the -N flag) also allow passing multiple arguments. However, there are some data structures that can never be passed. E.g., passing pointers containing a graph is in general not a good idea. The full language can be found in the RFCs, and is reproduced at the end of this reading.

We can then run `rpcgen hello.x`, which generates the common header file `hello.h`, the client stub file `hello_clnt.c` and the server stub file `hello_svc.c`. Both the latter files should be compiled separately, using `gcc -c hello_clnt.c` and `gcc -c hello_svc.c` respectively. Then one can write the service implementation. If desired, the command `rpcgen -Ss hello.x` (“Sample server”) can be used to produce a template that one can modify. We write it in `hello_svc_impl.c` like this:

```
#include <stdio.h> /* added so that we can use printf */
#include "hello.h"

int *printmsg_1_svc(char **argp, struct svc_req *rqstp) {
    static int result;
    printf("Got %s\n", *argp); /* added to print the message */
    result = 0;                /* added to return some value to the client */
    return &result;
}
```

The name of this function is fixed: “printmsg” is the name of the service converted to lower case, the “1” is its version number, and the “svc” indicates that it is the implementation of the server (the last “_svc” portion is not added in Sun’s implementation). We modify it so that the argument string is printed on the console. We then compile and link it with the server stub using `gcc hello_svc_impl.c hello_svc.o`. Once that is done, the program can be executed, providing the service to everybody who can connect to the computer that runs the program.

Now we can write clients to request for the service. Again, a template can be obtained, by running `rpcgen -Sc hello.x`. We can then modify it as desired:

```
#include <stdlib.h>
#include <stdio.h>
#include "hello.h"

void helloworld_1(char *host) {
```

```
CLIENT *clnt;
int *result_1;
char *printmsg_1_arg = "hello, world!"; /* added: always print this */
clnt = clnt_create (host, HELLOWORLD, HELLOWORLDVERS, "udp");
if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
}
result_1 = printmsg_1(&printmsg_1_arg, clnt);
if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
printf("Got %d\n", *result_1); /* added: show the result */
clnt_destroy (clnt);
}

int main (int argc, char *argv[]) {
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    helloworld_1 (host);
    exit (0);
}
```

The client code looks complicated, but in fact it only has three important calls. One is the `clnt_create()`, which finds the server, and makes the necessary connection. It returns you a `CLIENT*`, which is needed in every subsequent RPC. Once a program get a `CLIENT*`, it can make any number of RPCs, until it calls `clnt_destroy()`, destroying the connection. We call the `printmsg_1()` function, which is the client stub function created by `rpcgen`. The remaining unfamiliar functions are boring ones to print error messages when unexpected events occurs on client creation (`clnt_pcreateerror()`) and RPC (`clnt_perror()`). Once the program is ready, we compile the program, link it with the client stub file, and run it to obtain service from the server.

3. Data types

RPC allows the direct use of several data types: 4-byte **integers**, 8-byte **hyper** integers, 4-byte **floats**, 8-byte **doubles**, 1-bit **boolean** and NULL-terminated **strings** (of 0 to $2^{32}-1$ bytes). However, further data types can be defined and used as procedure arguments, using arrays, structures and unions. For example, a service returning an array of records, each containing a number and a string, can use the following specification in the `.x` file:

```
struct recordlist {
    int id;
    string name<>;
    char name2[16];
    recordlist *next;
};
typedef recordlist recordlist_array<1024>;
```

This defines two types `record` and `recordlist`. The `record` type is a structure containing an integer and a variable length string; the `recordlist` type is a variable length array of at most 1024 elements. A variable length array is specified by angle brackets (like `<>` or `<1024>`, the number indicating the maximum length, which defaults to $2^{32}-1$). A fixed length array is specified by square brackets (like `[10]`). An optional argument is represented by adding a `*` character, as in the `next` field above—it will be translated to a C pointer. Arrays and optional arguments are available only when defining new types, not when defining services (i.e., you cannot define a service to take a `string<>`).

When `rpcgen` is executed, marshaling function are generated to deal with these types. It is stored in its own file, ending with `_xdr.c`. The file should be compiled like the stub files, and should be linked to both the client and the server programs. The common header (`.h`) file contains the corresponding C-language types:

```
struct recordlist {
    int id;
    char *name;
    char name2[16];
    struct recordlist *next;
};
typedef struct recordlist recordlist;

typedef struct {
    u_int recordlist_array_len;
    recordlist *recordlist_array_val;
} recordlist_array;
```

The `record` type is as we would expect. The `recordlist` type, being of variable length, contains an extra field `recordlist_len`, which specifies how many elements are in the array `recordlist_val`.

4. Further information

Functions related to RPC and XDR are documented in the `rpc(3)` and `xdr(3)` (or `rpc(3nsl)` and `xdr(3nsl)`) man pages. The RFCs mentioned above can be consulted to find the exact external representation of the data types. The source of the reading contains a “hello” directory which has the examples shown.

Appendix A. The RPC Language in BNF notation

specification:
 definition *

definition:
 program-def
 / *constant-def*
 / *type-def*

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*
 (, *type-specifier*) *) = *constant* ;

constant-def:
 const *identifier* = *constant* ;

type-def:
 typedef *declaration* ;
 / **enum** *identifier* *enum-body* ;
 / **struct** *identifier* *struct-body* ;
 / **union** *identifier* *union-body* ;

declaration:
 type-specifier *identifier*
 / *type-specifier* *identifier* [*value*]
 / *type-specifier* *identifier* < [*value*] >
 / **opaque** *identifier* [*value*]
 / **opaque** *identifier* < [*value*] >
 / **string** *identifier* < [*value*] >
 / *type-specifier* * *identifier*
 / **void**

value:
 constant
 / *identifier*

type-specifier:
 [**unsigned**] **int**
 / [**unsigned**] **hyper**
 / **float**
 / **double**
 / **bool**
 / *enum-type-spec*
 / *struct-type-spec*
 / *union-type-spec*
 / *identifier*

enum-type-spec:
 enum *enum-body*

enum-body:
 {
 (*identifier* = *value*)
 (, *identifier* = *value*) *
 }

struct-type-spec:
 struct *struct-body*

struct-body:
 {
 (*declaration* ;)
 (*declaration* ;) *
 }

union-type-spec:
 union *union-body*

union-body:
 switch (*declaration*) {
 (**case** *value* : *declaration* ;)
 (**case** *value* : *declaration* ;) *
 [**default** : *declaration* ;]
 }