

CSIS0234B Computer and Communication Networks (Class B)

Reading for Tutorial 4

Introducing RPC (Remote Procedure Calls)

Sockets allow programmers to write programs that communicate through the network, using bytes or datagrams containing bytes as unit of transfer. It provides a low-level interface to the networking subsystem of the OS. But this is very unfriendly to programmers who want to communicate larger, variable sized data structures. As we see in assignment 1, many network programs end up with a lot of code to break up the byte stream into application messages, differentiate various kinds of messages, determining how variable length strings or arrays should be stored, converting between byte orderings, etc.

As usual, no programmer wants to spend time for uninteresting repeated tasks, so libraries are built so that the problem can be solved once and for all. Using these libraries, the programmer writes a description about what services a server provides, what parameters it requires, etc. The library then uses a standardized way to perform the network communication. Programmers typically do not need to know how the communication is achieved. There are many such libraries, and we will look at one particularly simple one called RPC (Remote Procedure Call). It is the basis on which the Network Filesystem (NFS) is built upon.

1. The abstraction of RPC

Every programmer knows what happens when a function is called: the caller prepares arguments, calls a 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. 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 all the “**stub 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 contain the stub functions for the server and the client respectively. The server stub file also contains a `main()` function, which binds itself to ports waiting for clients to connect to it.¹

Then the programmer writes functions which actually **implements the declared services**, and link it with the server stub file to form a full executable program. He can then write client programs that **calls the client stub**, and link it with the client stub. The client can then obtain network service using function calls.

¹Interestingly, RPC servers do not use well-known ports. Instead, an arbitrary port is chosen by the OS, and the server registers the port and the services to a standard **portmapper** service, which is constantly running (the process named `portmap`). The portmapper itself is running at fixed ports (TCP/UDP port 111). When the client wants to get serviced, the portmapper is consulted to find the correct port to send requests to.

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 value of the program 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 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_printmsg.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_printmsg.c hello_svc.o`. Once that is done, the program can be executed, providing the service to everybody who can connect to the computer running 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> /* added for exit() */
```

```
#include <stdio.h> /* added for printf() */
#include "hello.h"

void helloworld_1(char *host) {
    int *result_1;
    char *printmsg_1_arg = "Hello, world!"; /* added: always print this */
    CLIENT *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");
    } else {
        printf("Got %d\n", *result_1); /* added: get 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);
    return 0;
}
```

Then one can compile the program, linking it with the client stub, and run it to obtain service from the server. The client code looks quite complicated, but in fact it only has three important calls. One is the `clnt_create()` library call, which establishes finds where is server, and makes a connection is necessary. It returns you a `CLIENT*`, which is needed in every subsequent remote procedure call. Once you get a `CLIENT*`, any number of RPC calls can be made with it, until it calls `clnt_destroy()`, destroying the connection. We make only one, by using 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 calling (`clnt_perror()`).

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 both an array of records, each containing a number and a string, it defines the following in the specification (.x) file:

```
struct record {
    int id;
    string name<>;
};
typedef record recordlist<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 `<>`), while a fixed length array is specified by square brackets (like `[10]`).

When `rpcgen` is executed, marshaling functions 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 program. The common header (`.h`) file contains the corresponding C-language types:

```
struct record {
    int id;
    char *name;
};
typedef struct record record;

typedef struct {
    u_int recordlist_len;
    record *recordlist_val;
} recordlist;
```

The `record` type is as we would expect. The `recordlist` type 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* ;]
 }