Lecture 9: Remote Procedure Call

References for Lecture 9:

1) Unix Network Programming, W.R. Stevens, 1990,Prentice-Hall, Chapter 18.
2) Unix Network Programming, W.R. Stevens, 1998,Prentice-Hall, Volume 2.

Questions:

1) What is the difference between procedure and function?
2) What is definition of RPC?
3) Why do we need RPC?
4) How is RPC different from LPC? Or what is the execution order of RPC?
5) What is the purpose of stub?
6) How to send a request and how to get the response?
7) How to write and generate RPC programs?

Definition and Why?

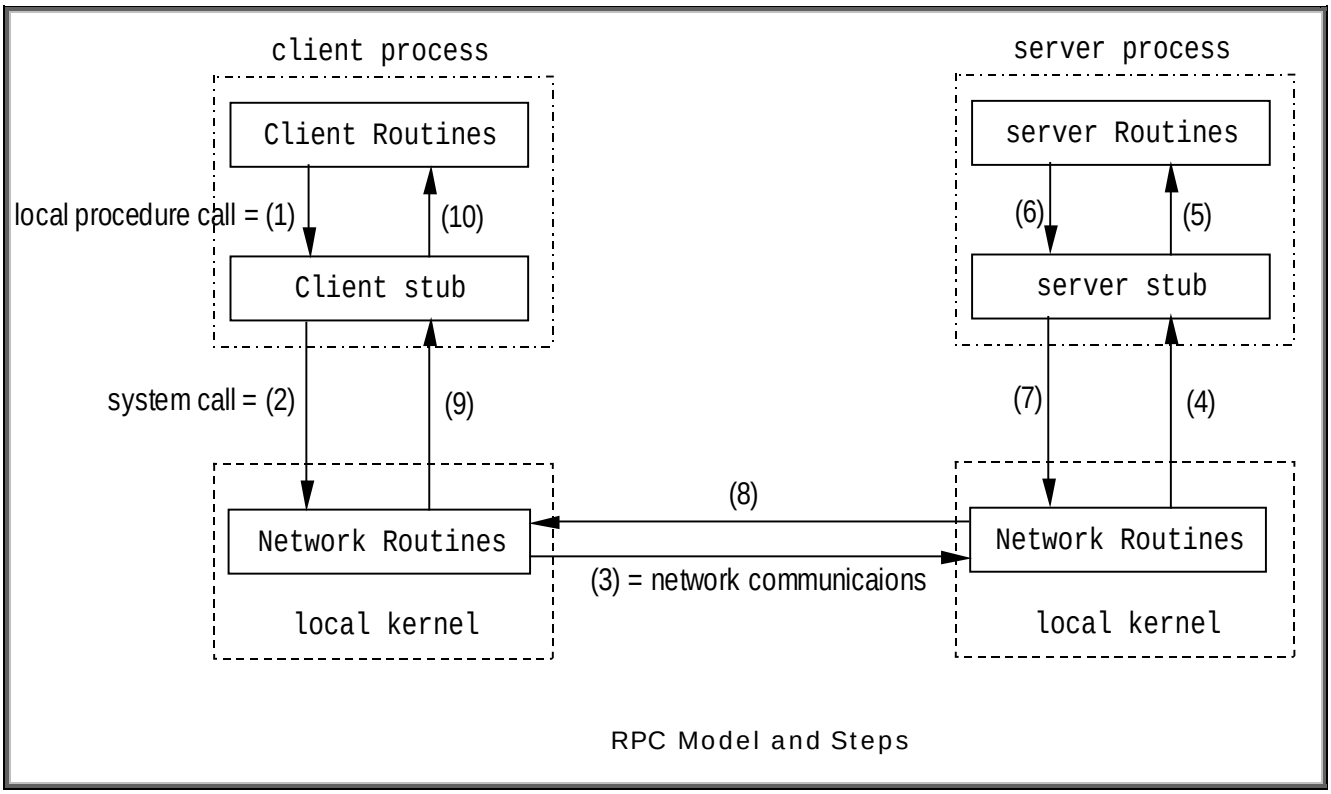
Remote Procedure Call is used to execute subroutines on other hosts. The ideas is to provide a simple mechanism for distributed computing without the need to include socket code (or something similar) in the program.

RPC hides all the network code into the stub procedures. This prevents the application programs, the client and the server, from having to worry about details such as sockets, network byte order, and the like.

Many systems supports remote procedure calls. We will discuss only the Sun RPC implementation.

Execution Steps

The actual implementation is much more complicated than making a local procedure call. A client process sends a request to execute the remote procedure. The server process then executes the remote procedure and sends a response to the client in order to return the result. The actual steps are:



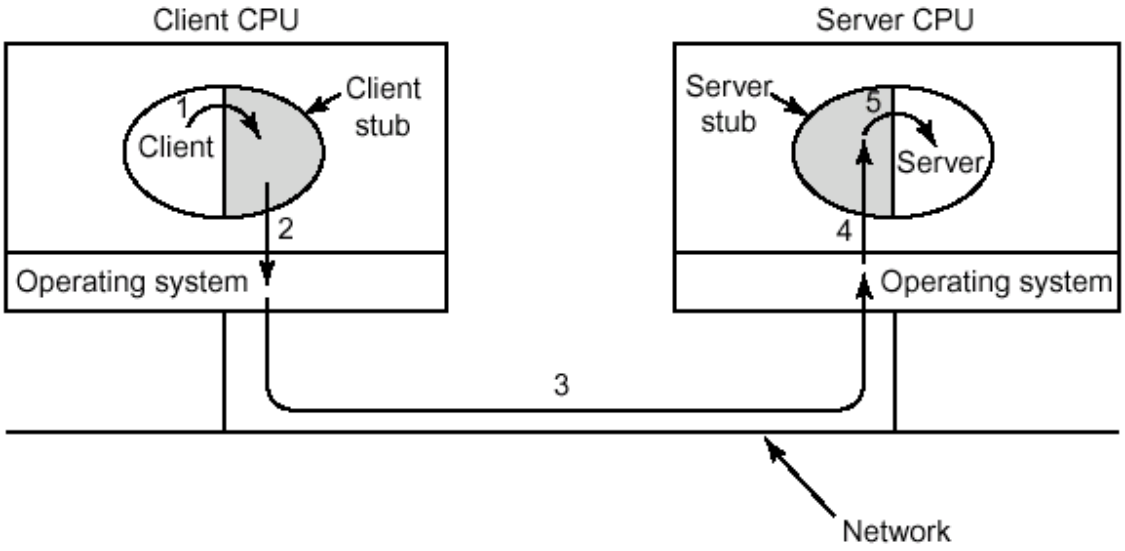
RPC Model and Steps

Note how the client and server routines use stub programs for the network communication. The RPC routine hides all the details. This simplifies the creation of the program and isolates the network code. Before these steps can be performed, the server must be started on the appropriate host. The specific steps are:

(1) The **client** calls the procedure. It calls the client stub, which packages the arguments into network messages. Packaging is called **marshaling.**
(2) The **client stub** executes a system call (usually write or sendto) into the local kernel to send the message.
(3) The **kernel** uses the network routines (TCP or UDP) to send the network message to the remote host. Note that connection-oriented or connectionless protocols can be used.
(4) A **server stub** receives the messages from the kernel and **unmarshals** the arguments.
(5) The **server stub** executes a local procedure call to start the function and pass the parameters.
(6) The **server process** executes the procedure and returns the result to the server stub.
(7) The **server stub** marshals the results into network messages and passes them to the kernel.
(8) The **messages** are sent across the network.
(9) The **client stub** receives the network messages containing the results from the kernel using *read* or *recv*.
(10) The **client stub** unmarshals the results and passes them to the client routine.

Stub: representative for communication, hiding socket code

Another simplied figure for RPC steps:



How to be connected to the server process:

In order to communicate between the server and the client, the client needs to know the IP address and port number of the server. The IP address can be determined from the host name. The port number is determined by contacting the port mapper on that host. On Solaris, the port mapper is called **rpcbind**. **rpcbind** uses the well-known address of 111. The connection between the client and server is established as follows:

(1) when the server program is started, it creates a socket and binds an ephemeral port to the socket. The server then registers its program number and version number with the port mapper. After this, the server waits for a client request.

(2) The client process call **clnt_create( )**. One of the parameters is the hostname, which is used to contact the port mapper on that host. The port mapper daemon finds the matching program number and the version number and returns the ephemeral port of the server to the client.

(3) The client uses this port number to communicate directly with the RPC server. Whether using TCP or UDP, the client sends the request multiple times if no response is received.

RPC is normally considered to be at the presentation layer (layer 6). One of the things it does is to provide the data conversion between hosts, so different machine types can be used. Sun RPC uses the extended Data Representation (XDR) standard to format data. Both TCP and UDP are supported.

Transparency Issues:

The system needs to provide a transparent interface for the client, so that there is no distinction between making a remote procedure call and making a local function call. The client and server stubs hide the network code, but there are other issues that need to be addressed:

Parameter Passing – can't pass parameters by reference, since the subroutine and the calling program don't share the same address space. Sun RPC allows only a single argument and a single result. A structure is required for multiple values.

Binding – the client needs some way to determine which host is a server. Choices are to require that the client knows which host to contact, or use s superserver that keeps track of the addresses of each server, or use a centralized database where each host indicates which servers it is willing to run.

Sun RPC takes the following approach. The port mapper on the remote host is contacted for the port number. The port mapper also accepts the broadcast requests. If a matching server is found, the request is passed to the server. The port number is then returned with the results, so the client can be connected directly to the server on future calls.

Transport Protocol – Sun RPC supports TCP and UDP. TCP is a byte-stream protocol, so there are no message delimiters. To solve this, a 32-bit integer giving the number of bytes is placed at the beginning of each record. With UDP on older systems, at most 8192 bytes can be sent for the arguments or results of one call. The maximum can never exceed the size of a UDP datagram, which is 64 K – headers.

Exception Handling – not only could the typical errors, such as segmentation fault, occur in the remote procedure, but also network problems are also possible. A timeout is usually used to detect server crashes. The client might also wish to terminate the server. With Sun RPC, the client cannot send an interrupt to the server. Both UDP and TCP handle timeouts and retransmissions. UDP will terminate after some number of unsuccessful attempts.

Call Semantics – because of network problems, the request to start a remote procedure might be sent multiple times. Procedures that can be executed multiple times without a problem are called idempotent. Examples include computing a square root or checking an account balance. There are three choices of call semantics:

- Exactly once – hard to achieve. Examples include saving money.
- At most once – a normal return means it is executed. An error return, we don't know. Examples include take money
- At least once – Ensure it isexecuted. Used for idempotent functions. Repeat request until successful. May execute multiple times. Examples include inquiring the account.

Sun RPC supports call semantics by assigning a unique transaction ID (xid) to every client request. The RPC server can test this value for equality, but cannot change it. TCP and UDP initialize the xid to a random value when the client handle is created. The value is returned with the result and used to make sure the response goes to the right client. The UDP server functions can cache responses to keep track of which requests have been received. If a duplicate request is received, the first response (not the new one) is returned to the client. The idea is to provide at-most-once semantics with UDP.

Data Representation – Need a standard representation, so the client and server can execute on different architectures. Sun RPC uses the XDR data representation standard. Integer: big-endian or litter-endian. Char: EBCDIC or ADCII. Floating: IEEE starndard?

Performance – there can be considerable overhead for calling an RPC. For example, the overhead might be 100 times the overhead of a local procedure call. Sun RPC uses several mechanisms, such as passing pointers, to minimize coping data. The purpose of RPC is to simplify network programming, not just to replace LPC with RPC. Reasons for RPC: 1) not supported locally such as supercomputer and VLDB, 2) even for efficiency, |computation| >>|communication|.

Security – May need to restrict who can execute a program on the server. Sun RPC supports several security options. The default is **null authentication**. If UNIX authentication is chosen, then the time stamp, host ID, user ID, and group ID are sent to the server with each request. The server can check these fields to decide whether or not to process the clients' request (Weak security). DES and Kerberos authentication are also valid options.

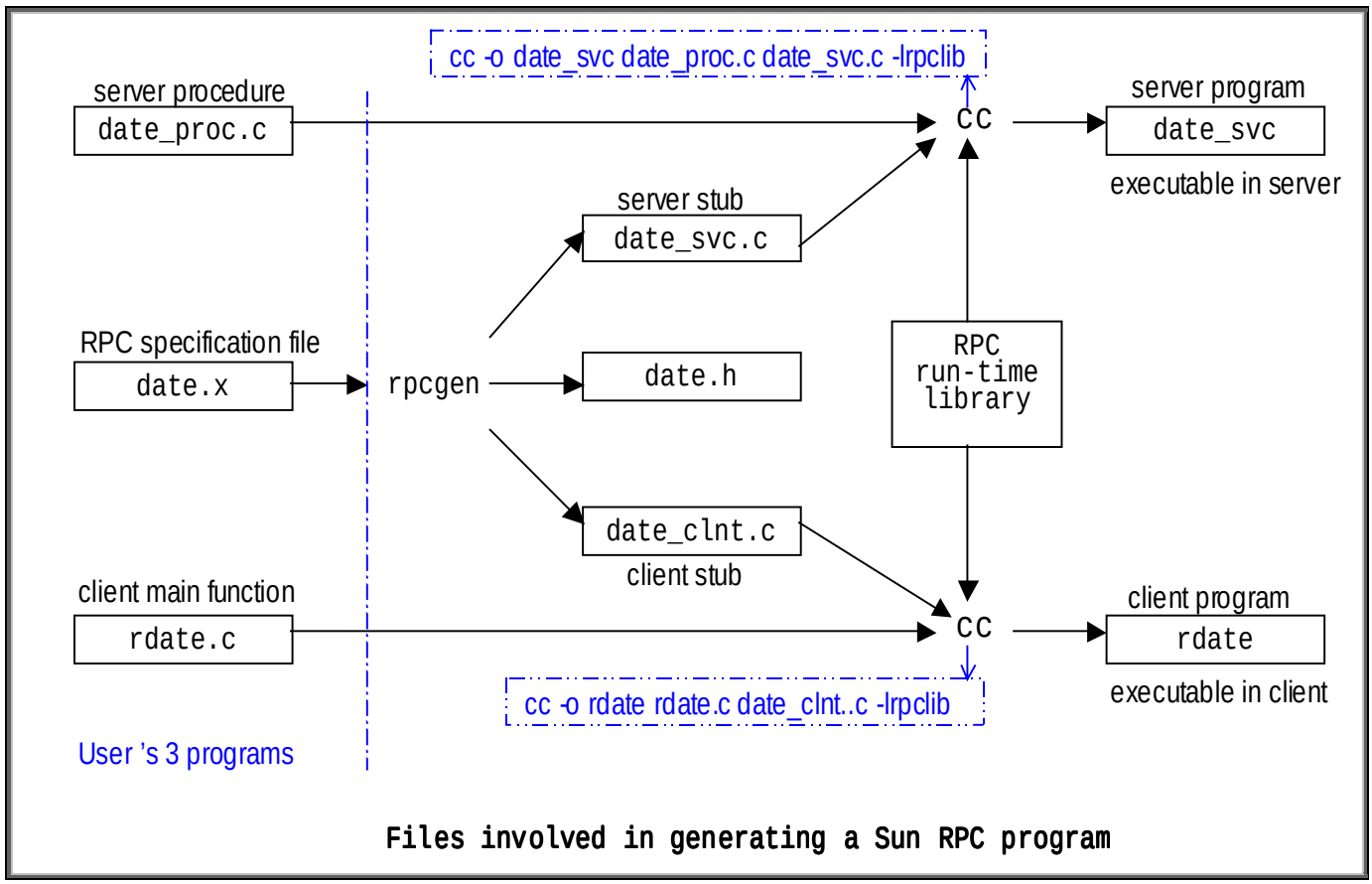
Questions:
1) Since the overhead of RPC might be 100 times the overhead of a local procedure call, why do we still need RPC? 2) Is a procedure that appends something to a file idempotent? 3)Why can RPC not pass parameters by reference? 4) Explain how Sun RPC maintains at-most-once semantics?

How to write RPC programs

Three files are required for a remote procedure: The RPC specification file, the client code, and the sever code.

Sun RPC has three components:

- rpcgen — a compiler that generates the client and server stub for the definition of a remote procedure interface. Use –C to generate ANSI C code.
- XDR — eXternal Date Representation. Used to encode the data into a portable format. Simplifies execution among different computer architectures.
- A runtime library; -lrpclib



Files involved in generating a Sun RPC program

Example:

Server: date_svc & /* This is a time server */

Client: rdate gigastar

Client: rdate.c to call 2 remote functions on gigastar

Epoch

lresult=bin_date(); /* number of seconds since **00:00:00,Jan.1,1970 GMT** */

sresult=stri-date(lresult); /* 26-character ASCII string fit for human consumption such as **Mon Dec 17 16:30:00 2001***/

date.x

```
/* date.x - Specification of remote date and time service. */

/*    Define 2 procedures:
 *    bin_date_1() returns the binary time and date (no arguments).
 *    str_date_1() takes a binary time and returns a human-readable string.
 */

program DATE_PROG {
        version DATE_VERS {
       long      BIN_DATE(void) = 1;    /* procedure number = 1 */
       string    STR_DATE(long) = 2;    /* procedure number = 2 */
        } = 1;                    /* version number = 1 */
} = 0x31234567;                   /* program number = 0x31234567 */
```

%rpcgen date.x ➔ date.h

```
/*      Please do not edit this file. It was generated using rpcgen. */

#include <rpc/rpc.h>
#define  DATE_PROG    0x31234567
#define  DATE_VERS    1
#define  BIN_DATE       1
extern    long    * bin_date_1( );
#define  STR_DATE       2
extern    char    ** str_date_1( );
extern int date_prog_1_freeresult( );
```

Notes:1) The program numbers are 32-bit integers. The user-defined program number should lie between **0x20000000 – 0x3fffffff**.

2) rpcgen converts the remote procedure names **BIN_DATE** and **STR_DATE** as **bin_date_1** and **str_date_1**.

3) Sun RPC specifies that the remote procedure return the **address** of its return value.

4) Only **one argument and one result** is permitted, so a structure must be used to pass/return multiple values.

5) Please also have a look at date_clnt.c and date_svc.c, which are client stub and server stub.

Questions: 1) What is the purpose of program number and version number ?

2) How to avoid the same program number and version number from different users?

rdate.c

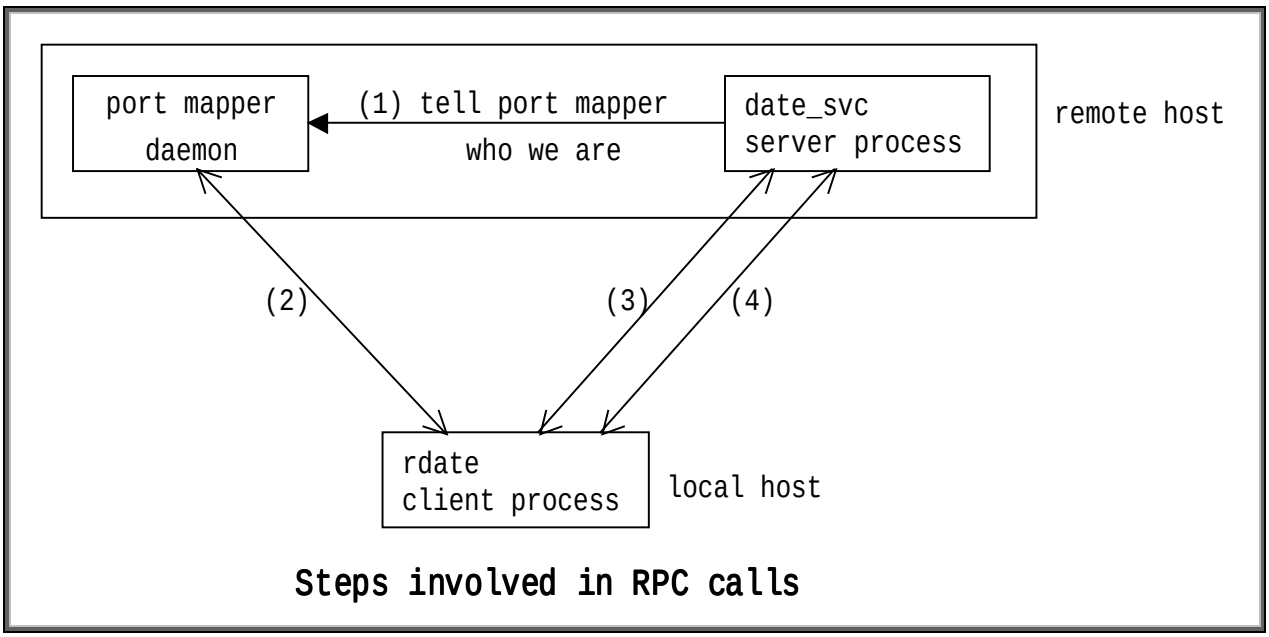
```
/* rdate.c - client program for remote date service. */

#include      <stdio.h>
#include      <rpc/rpc.h>       /* standard RPC include file */
#include      "date.h"          /* this file is generated by rpcgen */

main(argc, argv)
int  argc;
char      *argv[];
{
      CLIENT          *cl;        /* RPC handle */
      char            *server;
      long            *lresult;   /* return value from bin_date_1() */
      char            **sresult;  /* return value from str_date_1() */

      if (argc != 2) { fprintf(stderr, "usage: %s hostname\n", argv[0]);         exit(1);  }
      server = argv[1];

      /*Create the client "handle." */      Program number    Version number

      if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
            /*    Couldn't establish connection with server. */
            clnt_pcreateerror(server);
            exit(2);
      }
       /*    First call the remote procedure "bin_date".    */
                              argume          RPC handle
      if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
            clnt_perror(cl, server);
            exit(3);
      }
      printf("time on host %s = %ld\n", server, *lresult);

      /*    Now call the remote procedure "str_date".    */
      if ( (sresult = str_date_1(lresult, cl)) == NULL) {
            clnt_perror(cl, server);
            exit(4);
      }
      printf("time on host %s = %s", server, *sresult);
      clnt_destroy(cl);       /* done with the handle */
      exit(0);
}
```

dateproc.c

```
/*    dateproc.c - remote procedures; called by server stub.    */

#include      <rpc/rpc.h> /* standard RPC include file */
#include      "date.h"      /* this file is generated by rpcgen */

/*    Return the binary date and time.    */

long * bin_date_1( )
{
     static long    timeval;          /* must be static */
     long           time();           /* Unix system call */

     timeval = time((long *) 0);    /* returns time in seconds since 00:00:00, January 1, 1970 */

     return(&timeval);
}

/* Convert a binary time and return a human readable string. */

char **str_date_1(bintime)
long      *bintime;
{
     static char    *ptr;             /* must be static */
     char           *ctime();         /*c standard function */

     ptr = ctime(bintime);          /* convert to local time */

     return(&ptr);                  /* return the address of pointer */
}
```

Questions: 1) Why is there no main function in dataproc.c ?

Steps in RPC calls:



(1) When the server process is started, it creates a socket and binds an ephemeral port to the socket, then call **svc_register( )** to registers its program number, version number to the port mapper. After this, the server waits for a client request. **Step 1 is done automatically by server stub.**

(2) When the client process is started, it calls **clnt_create()**. One of the arguments is the hostname, which is used to contact the port mapper on that host. The port mapper daemon finds the matching program number and version number and returns the ephemeral port of the server to the client.

(3) The cleint first calls **bin_date_1()** function. This function, defined in the client stub, uses the port number obtained in step (2) to communicate directly with RPC sever. Whether using TCP or UDP, the client stub sends the request multiple times if no response is received. When server stub receives the request, it determines which procedure should be called. After bin_date_1() is called, the server stub converts the result into XDR format, package it into a datagram for sending back to the client stub. The client stub converts the result as required and returns it to our client program.

(4) The client then calls **str_date_1()** function, following the same way as step(3).

Oh! As a user, I can use RPC as almost easily as if I am using LPC. However, my convenience results from the fact that system or stubs help me do plenty of work automatcially.

Answers to Questions:

1) What is definition of RPC?

2) Why do we need RPC?

: To hide the network programming complexity for the convenience of distributed computing.

3) How is RPC different from LPC? Or what is the execution order of RPC?

: See execution steps and transparency issues. As a result, RPC is much more complicated than LPC.

4) What is the purpose of stub?

: stub hides socket code inside and behave as an agent for communication.

5) How to send a request and how to get the response?

: First, use library function **clnt_create**(server, DATE_PROG, DATE_VERS, "udp") to create a CLIENT handle.

: Second, call remote functions in almost the same way as if you are calling local functions.

: The request and response will be automatically processed by client stub and server stub.

6) How to write and generate RPC programs?

: write programs as usual,

: write a specification file.