Chapter 22/23

Building Remote Procedure Calls

Problem: RPC headers are complicated (but repetitive)

Caller: Build message, call service with correct program numbers, specify where arguments come from/go to, generate authentication, wait for reply.

Server: Need to register service, accept calls (including authentication), handle arguments, build reply, send reply

Problem: The arguments have to be correctly XDRd, the XDR routine must match the argument types.

Solution: Automate the processes. (RPC compiler)
Automatically generate as much of the code as possible.
Given the argument types, generate the XDR routines.

Stub procedures: normal procedure.
Stub does all rpc handling,
Effect: Stub hides RPC complexities

Good programming style: Hide all this stuff.

Solution: build interface procedures that hide the stub procedures.

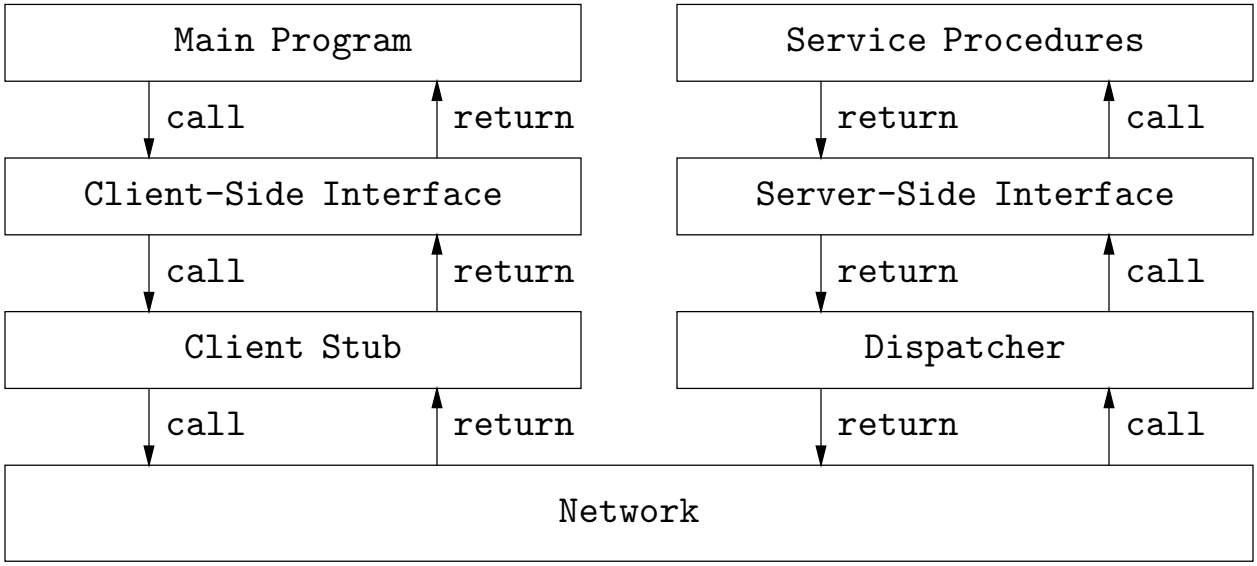
Client-Side Stubs and Interfaces

Client interface makes calls look like original procedure. (Hides peculiarities of stub from main program)

Client stub has procedure name similar to server stub.

Stub builds the XDR stream, authentication and other parts of the call.

Stub Connects to the remote procedure.



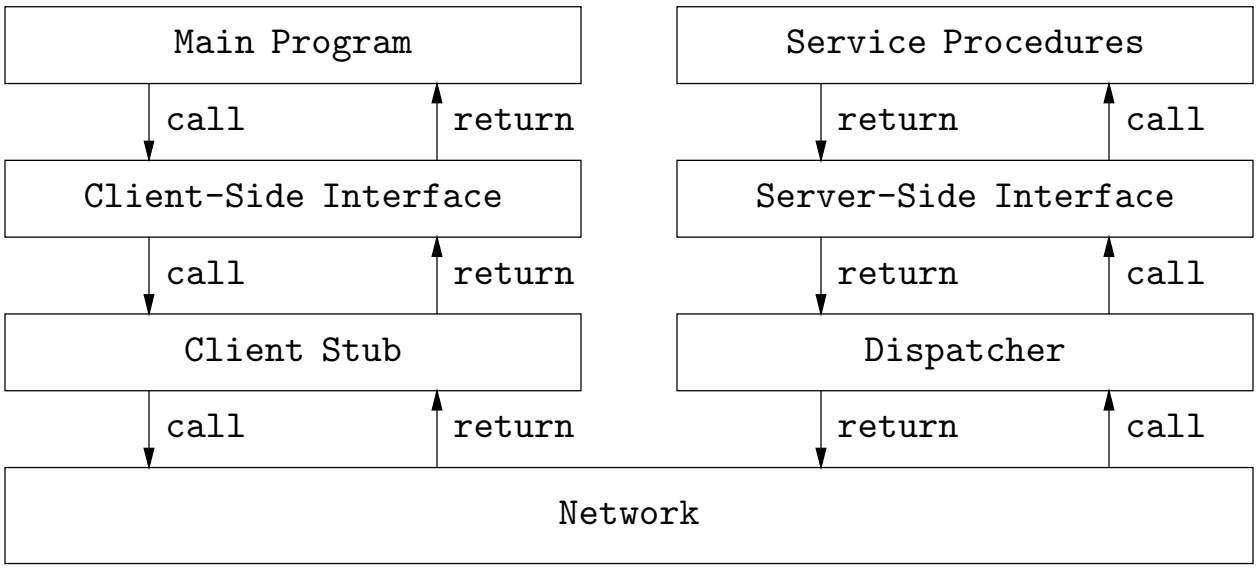
The more complicated separation will be used so parts can be generated by software.

Server-Side Dispatcher and Interfaces

Call arrives at dispatcher.

Dispatcher examines procedure number and
calls correct server-side interface procedure.

Server interface calls real procedure.
(Hides peculiarities of dispatcher from service procedures)

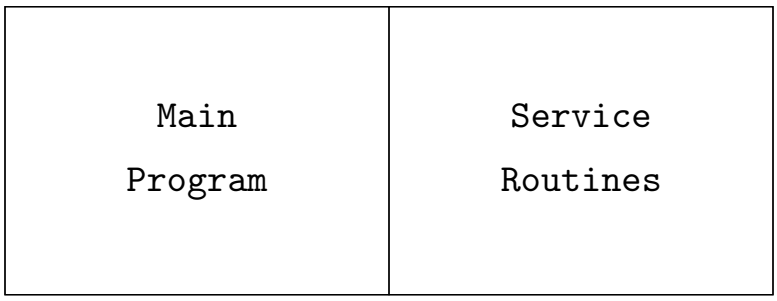


Using "interfaces" the main program needs a couple lines added and the real procedures don't need to be changed.

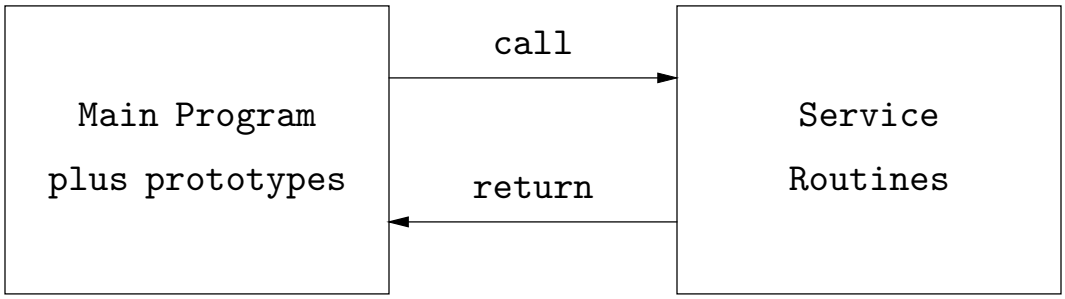
Methodology

How to Build RPC Programs

1) Build a well modularized program



2) Split the program. Add prototypes for the service routines to the main program.

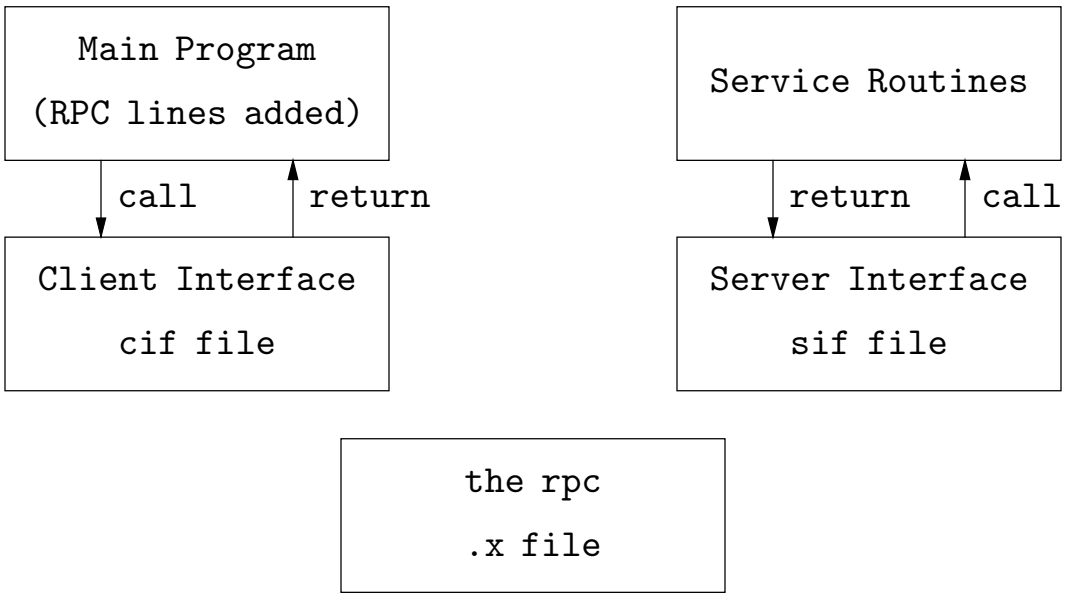


Build RPC Stuff

3) Build the `.x` file.
This defines exactly what will be passed over the network.

4) Add the necessary lines to the main program

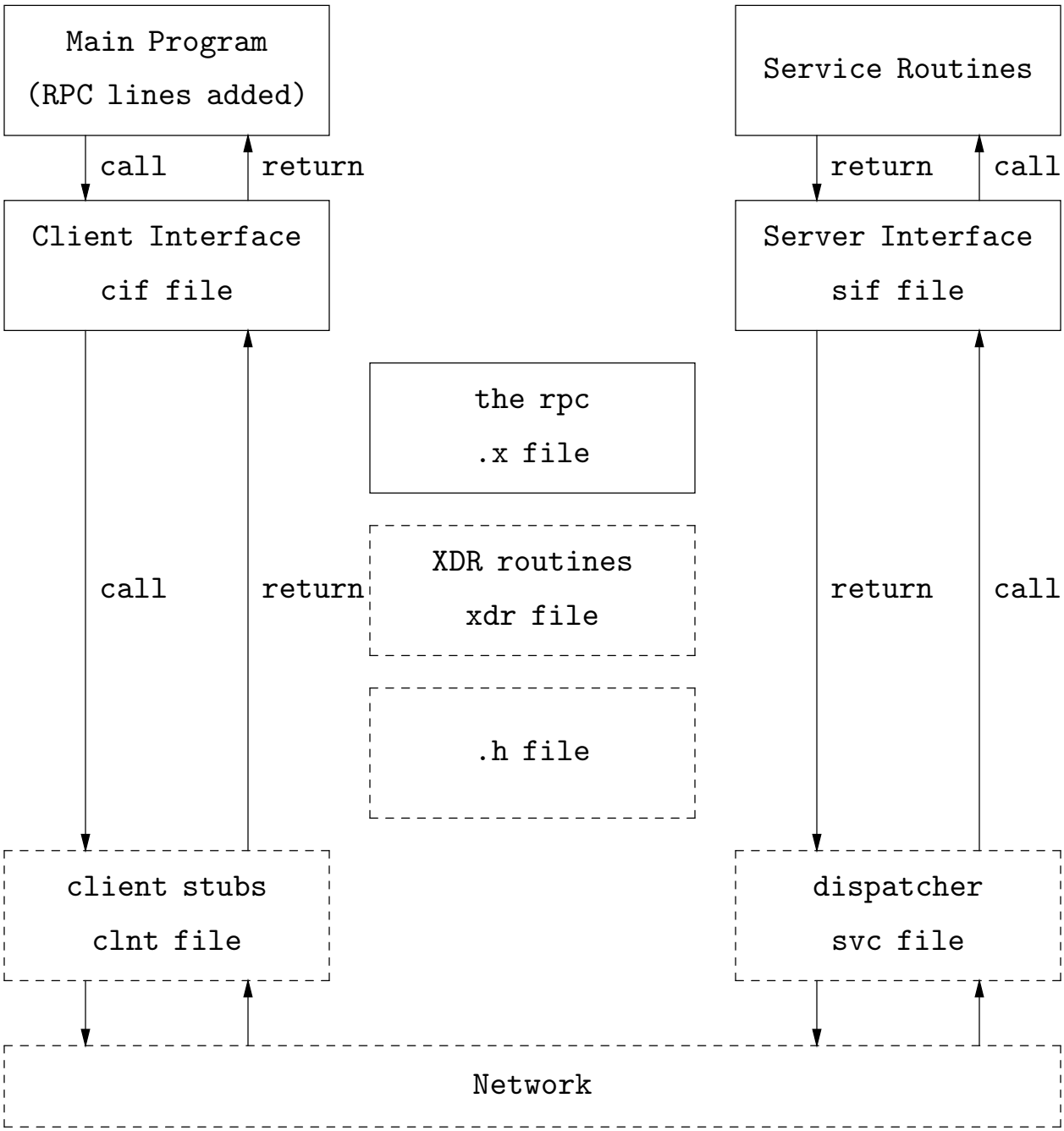
5) Build the Client and Server interfaces.



Apply rpcgen

This generates the stubs, the dispatcher, the XDR routines and the header (`.h`) file.

The dispatcher and server stubs are placed in the svc file.



Split Main Program with Prototypes

```
/* File: convert_client.c */
/* Convert a sentence to upper or lower case */
void upper_it(char *sentence);
void lower_it(char *sentence);
int main() {
  char buffer[100]; /* holds the sentence */
  char whichway[5]; /* to upper or to lower */
  /* control-C to exit */
  for (;;) {
    printf("enter the string ");
    gets(buffer);
    printf("u) to upper, l) to lower? ");
    gets(whichway);
    switch (whichway[0]) {
      case 'u':
        upper_it(buffer);
        break;
      case 'l':
        lower_it(buffer);
        break;
      default:
        printf("unrecognized option\n");
    }
    printf("%s\n", buffer);
  }
}
```

Original Service Procedures

```
/* File: convert_services.c */
#include <ctype.h>  /* toupper, tolower*/
/* Convert a sentence to upper or lower case */
void upper_it(char *sentence){
  char *letter = sentence;
  while (*letter) {
    *letter = toupper(*letter);
    letter++;
  }
}
void lower_it(char *sentence){
  char *letter = sentence;
  while (*letter) {
    *letter = tolower(*letter);
    letter++;
  }
}
```

Main program and service procedures nicely separated

Building the rpcgen File

convert.x:

```
/* convert.x */
%#include "Anything.h"
program CONVERTPROG {
  version CONVERTVER {
    string UPPER_IT(string) = 1;  /* procedure 1 */
    string LOWER_IT(string) = 2;  /* procedure 2 */
  } = 1;                          /* version number */
} = 0x20000064;                   /* program number */
```

The user gives the program number, the version number, the parameter lists and return types of the RPCs and the RPC numbers. % passes the `include` to the `convert.h` file.

Example (`LOWER_IT`): Creates `lower_it_1` as procedure number 2. It will take a string and return a string. The `_1` will match the version

This file is used by rpcgen to build the following files:

`convert_svc.c` – dispatcher
`convert_clnt.c` – client stubs
`convert_xdr.c` – any special xdr routines required by parameters or return values. In this example none are required so the file is not created.
`convert.h` – the header file

Header File

This file (`convert.h`) will be created when rpcgen is run.

```
#include "Anything.h"
#define CONVERTPROG ((u_long)0x20000064)
#define CONVERTVER ((u_long)1)
#define UPPER_IT ((u_long)1)
extern char **upper_it_1(char **, CLIENT *);
extern char **upper_it_1_svc(char **,
                             struct svc_req *);
#define LOWER_IT ((u_long)2)
extern char **lower_it_1(char **, CLIENT *);
extern char **lower_it_1_svc(char **,
                             struct svc_req *);
```

Defines the program number.
Defines the procedure numbers.
Names the procedures (`extern`)
`upper_it_1` and `lower_it_1` are called by the client-side interface.
`upper_it_1_svc` and `lower_it_1_svc` are what the server (dispatcher) calls.

Note that `UPPER_IT` gets "defined" as a 1;
don't use lower case or else `upper_it` will get "defined";
this definition will conflict with the name of your procedure (also called `upper_it`) and create a syntax error.

Main Program–RPC Version

```
/* Convert a sentence to upper or lower case */
#include <rpc/rpc.h>
#include "convert.h"
CLIENT *handle;
void upper_it(char *sentence);
void lower_it(char *sentence);
main() {
  char buffer[100]; /* holds the sentence */
  char whichway[5]; /* to upper or to lower */
  if ((handle = clnt_create("cheetah",
   CONVERTPROG, CONVERTVER, "tcp")) == NULL) {
    /* print the error message */
    clnt_pcreateerror("convert client");
    exit(-1);
  }
  /* control-C to exit */
  ... same from hear on
}
```

The only things added are the includes, the `handle` variable and the call to `clnt_create`.

The `clnt_create` initializes the rpc's to point to the convert server program on cheetah.

Note: Service routines are not changed at all.

A More Flexible Main Program

Our main always looked for the server on cheetah.

It would be better if it allowed a command line argument to specify the host and it defaulted to localhost.

This can be done by adding some of Comer's code:

```
char *host = "localhost"
switch (argc) {
case 2: host = argv[1];
...
if (( handle = clnt_create(host, ...
```

The Client-Side Interface (cif)

```
#include <rpc/rpc.h>
#include "convert.h"
/* convert_cif.c */
extern CLIENT *handle;
void upper_it(char *sentence){
  char **answer;
  answer = upper_it_1(&sentence, handle);
  strcpy(sentence, *answer);
}
void lower_it(char *sentence){
  char **answer;
  answer = lower_it_1(&sentence, handle);
  strcpy(sentence, *answer);
}
```

Interface to main program is the same.

Makes calls to the actual rpc.
Must pack arguments into the parameter structure and unpack them from the answer structure.

The Server-Side Interface (sif)

```
/* convert_sif.c */
#include <rpc/rpc.h>
#include "convert.h"
void upper_it(char *sentence);
void lower_it(char *sentence);
static char buffer[100];
static char *bufptr = buffer;
char **upper_it_1_svc(char **sentence,
                      struc svc_req *dummy){
  strcpy(buffer, *sentence);
  upper_it(buffer);
  return &bufptr;  /* yeilds &buffer */
}
char **lower_it_1_svc(char **sentence)
                      struc svc_req *dummy){
  strcpy(buffer, *sentence);
  lower_it(buffer);
  return &bufptr;  /* yeilds &buffer */
}
```

These procedures are called by the dispatcher.
They must unpack arguments from the parameter structure and pass them to the actual service routines.
They must pack answer into the answer structure.
The answer structure (`buffer`, `bufptr`) must be static.
Note: `&buffer` is not correct C.

Client Stub (created by rpcgen)

The file `convert_clnt.c`:

```
#include <rpc/rpc.h>
#include "convert.h"
/* Default timeout can be changed */
static struct timeval TIMEOUT = { 25, 0 };
char **upper_it_1(argp, clnt)
  char **argp;
  CLIENT *clnt;
{ static char *res;
  bzero(&res, sizeof(res));
  if (clnt_call(clnt, UPPER_IT, xdr_wrapstring,
    argp, xdr_wrapstring, &res, TIMEOUT)
     != RPC_SUCCESS) {
    return (NULL);
  }
  return (&res);
}
```

Lower it is same except for name change.

`clnt_call`: actual rpc call

`xdr_wrapstring`: xdr conversion for parameter

`argp`: parameter

`xdr_wrapstring`: xdr conversion for result

`res`: pointer to answer structure

Dispatcher (created by rpcgen)

Makes the `upper_it` and `lower_it` rpc's available.
Two parts, registration, actual dispatcher
`convert_svc.c`: registration portion (stripped)

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "convert.h"
static void convertprog_1(); /* dispatcher */
main()
{ SVCXPRT *transp;
  (void)pmap_unset(CONVERTPROG, CONVERTVER);
  /* create a udp service */
  transp = svcudp_create(RPC_ANYSOCK);
  /* register the udp version of the rpc */
  svc_register(transp, CONVERTPROG, CONVERTVER,
    convertprog_1, IPPROTO_UDP));
  /* create a tcp service */
  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  /* register the tdp version of the rpc */
  svc_register(transp, CONVERTPROG, CONVERTVER,
    convertprog_1, IPPROTO_TCP));
  /* run the dispatcher-dispatcher */
  svc_run();
}
```

Actual Dispatcher

Called for each RPC

```
static void convertprog_1(rqstp, transp)
{ a bunch of local variables including:
  char *(*local)();
  /* Set the function to call */
  switch (rqstp->rq_proc) {
  /* Direct call */
  case NULLPROC: (void)svc_sendreply(...
    return;
  case UPPER_IT:
    /* call set up, xdr to use, who to call */
    xdr_argument = xdr_wrapstring;
    xdr_result = xdr_wrapstring;
    local = (char *(*)(char*,
              struct svc_req *)) upper_it_1_svc;
    break;
  case LOWER_IT: -- similar ... }
  /* line up arguments */
  bzero(&argument, sizeof(argument));
  svc_getargs(transp, xdr_argument, &argument));
  /* call the procedure */
  result = (*local)(&argument, rqstp);
  /* send back the result */
  svc_sendreply(transp, xdr_result, result));
  /* clean up */
  svc_freeargs(transp, xdr_argument, &argument));}
```

Vector Program Example

Split Main Program with Prototypes

```
/* Vector operations interface */
#include "vectora.h"
void add(struct vector *a, struct vector *b,
   struct vector *c);
main() {
    struct vector p,q,r;
    p.x = 1;
    p.y = 4;
    q.x = 2;
    q.y = 5;
    add(&p, &q, &r);
    printf("answer (should be (3, 9)): %d, %d\n",
      r.x, r.y);
}
```

Service Routines

First two parameters are to be added,
result returned using the third parameter.
Structures are passed by address.

```
#include "vectora.h"
void add(struct vector *a, struct vector *b,
   struct vector *c) {
  c->x = a->x + b->x;
  c->y = a->y + b->y;
}
```

The rpcgen File

For illustration purposes we pass the parameters and return the result using arrays.

```
/* The structures used by the actual rpc calls. */
/* two vectors will be passed */
struct pass {
  int data[4];
  };
/* one vector will come back */
struct answer {
  int data[2];
  };
program VECTORPROG {
  version VECTORVER {
    answer ADD(pass) = 1;  /* procedure 1 */
  } = 1;                   /* version number */
} = 0x20000064;            /* program number */
```

Main Program–RPC Version

```
#include <rpc/rpc.h>
#include "vector.h"
CLIENT *handle;
#include "vectora.h"
void add(struct vector *a, struct vector *b,
   struct vector *c);
main() {
    struct vector p,q,r;
     if ((handle = clnt_create("cheetah",
      VECTORPROG, VECTORVER, "tcp")) == NULL) {
        /* couldn't connect to the server */
        clnt_pcreateerror("vector client");
        exit(-1);
    }
    p.x = 1;
    p.y = 4;
    q.x = 2;
    q.y = 5;
    add(&p, &q, &r);
    printf("answer (should be (3, 9)): %d, %d\n",
      r.x, r.y);
}
```

Only handle stuff and header lines are added.

The Client-Side Interface (cif)

Incorrect packing and unpacking will cause a program crash (segfault, or bus error).

```
#include <rpc/rpc.h>
#include "vectora.h"
#include "vector.h"
extern CLIENT *handle;
void add(struct vector *a, struct vector *b,
   struct vector *c) {
  struct pass params;
  struct answer *result;
  /* pack up the parameters a and b */
  params.data[0] = a->x;
  params.data[1] = a->y;
  params.data[2] = b->x;
  params.data[3] = b->y;
  /* call the remote function */
  result = add_1(&params, handle);
  /* unpack the answer into c */
  c->x = result->data[0];
  c->y = result->data[1];
}
```

One (pointer to a) parameter, one handle

The Server-Side Interface (sif)

If the answer is not packed into a static variable, it will be deallocated on return from the sif and the area pointed to will be reused leaving you with garbage.

```
#include <rpc/rpc.h>
#include "vectora.h"
#include "vector.h"
void add(struct vector *a, struct vector *b,
   struct vector *c);
answer *add_1_svc(pass *param,
                  struct svc_req *dummy) {
  struct vector p,q,r;
  static struct answer result;
  /* unpack the parameters */
  p.x = param->data[0];
  p.y = param->data[1];
  q.x = param->data[2];
  q.y = param->data[3];
  add(&p,&q,&r); /* call the actual procedure */
  /* pack up the answer */
  result.data[0] = r.x;
  result.data[1] = r.y;
  return &result; /* return the answer */
}
```

One (pointer to a) parameter, *one* (pointer to a) result.

xdr File (created by rpcgen)

rpcgen will create `vector_xdr.c`
this file has the xdr routines

```
#include <rpc/rpc.h>
#include "vector.h"
bool_t xdr_pass(xdrs, objp)
 XDR *xdrs;
 pass *objp;
{
 if (!xdr_vector(xdrs, (char *)objp->data, 4,
  sizeof(int), xdr_int)) {return (FALSE);}
 return (TRUE);
}
bool_t xdr_answer(xdrs, objp)
  XDR *xdrs;
  answer *objp;
{
  if (!xdr_vector(xdrs, (char *)objp->data, 2,
    sizeof(int), xdr_int)) {return (FALSE);}
  return (TRUE);
}
```

xdr the vector using by xdr-ing an int 4/2 times