RPC Programming with **`rpcgen`**

Issues:

- Protocol Definition File
- Client Programming
  - Creating an "RPC Handle" to a server
  - Calling client stubs
- Server Programming
  - Writing Remote Procedures

Protocol Definition File

- Description of the *interface* of the remote procedures.
  - Almost function prototypes
- Definition of any data structures used in the calls (argument types & return types)
- Can also include shared C code (shared by client and server).

XDR the language

- Remember that XDR data types are not C data types!
  - There is a *mapping* from XDR types to C types – that's most of what rpcgen does.
- Most of the XDR syntax is just like C
  - Arrays, strings are different.

XDR Arrays

- *Fixed Length* arrays look just like C code:

  `int foo[100]`

- *Variable Length* arrays look like this:

  `int foo<>` or **`int foo<MAXSIZE>`**

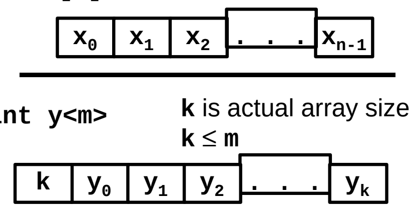
Implicit maximum size is 2^{32}-1

What gets sent on the network

`int x[n]`

`int y<m>`

`k` is actual array size
`k` $\leq$ **`m`**

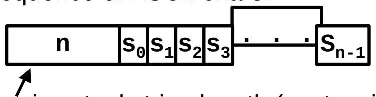


XDR String Type

- Look like variable length arrays:

  `string s<100>`

- What is sent: length followed by sequence of ASCII chars:



`n` is actual string length (sent as int)

Linked Lists!

```
struct foo {
  int x;
  foo *next;
}
```

`rpcgen` recognizes this as a linked list

The generated XDR filter uses **`xdr_pointer()`** to encode/decode the *stuff* pointed to by a pointer.

Check the online example "linkedlist".

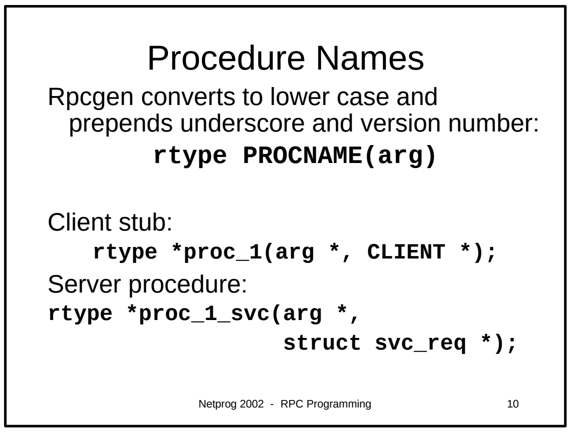
Declaring The Program

```
program SIMP_PROG {
   version SIMP_VERSION {
     type1 PROC1(operands1) = 1;
     type2 PROC2(operands2) = 2;
   } = 1;
} = 40000000;
```

Color Code:
Keywords Generated Symbolic Constants
Used to generate stub and procedure names

Procedure Numbers

- Procedure #0 is created for you automatically.
  - Start at procedure #1!
- Procedure #0 is a dummy procedure that can help debug things (sortof an RPC ping server).



Procedure Names

Rpcgen converts to lower case and prepends underscore and version number:

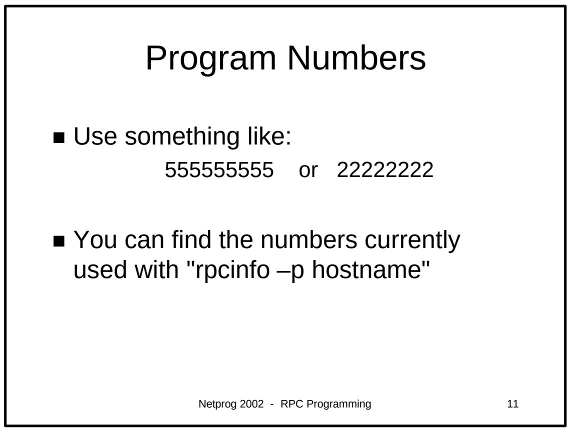
```
rtype PROCNAME(arg)
```

Client stub:

```
rtype *proc_1(arg *, CLIENT *);
```

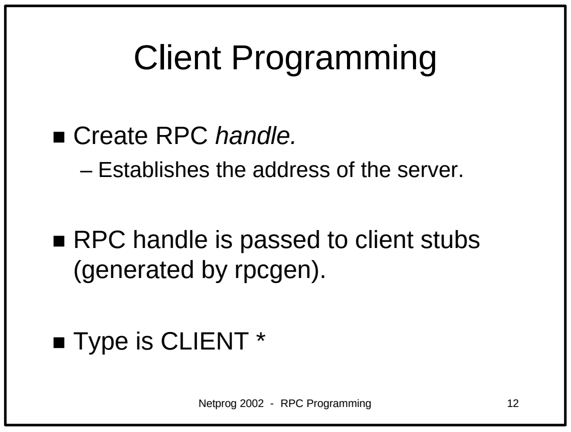
Server procedure:

```
rtype *proc_1_svc(arg *,
                  struct svc_req *);
```



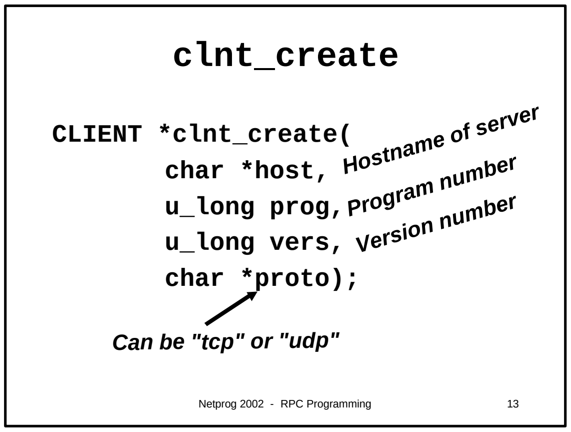
Program Numbers

- Use something like:
  555555555 or 22222222
- You can find the numbers currently used with "rpcinfo –p hostname"



Client Programming

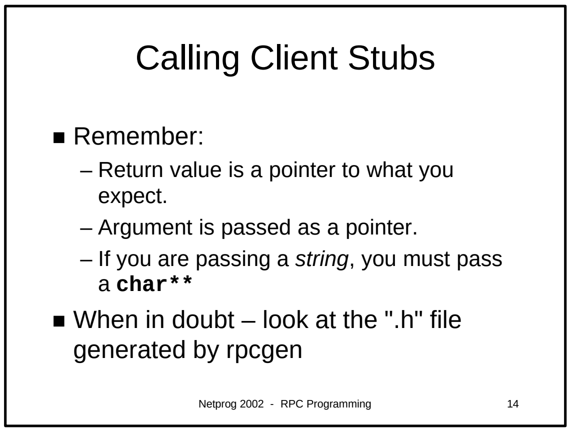
- Create RPC *handle.*
  – Establishes the address of the server.
- RPC handle is passed to client stubs (generated by rpcgen).
- Type is CLIENT *



clnt_create

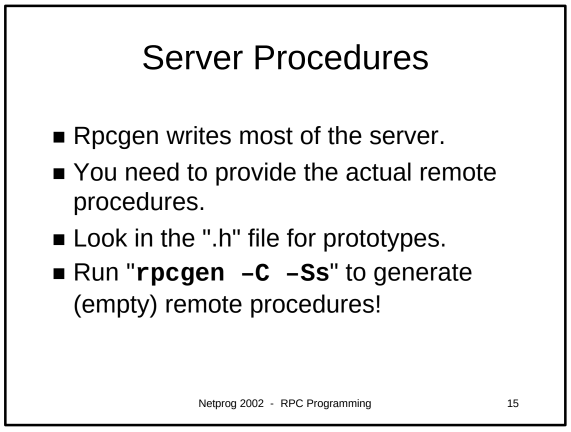
```
CLIENT *clnt_create(
       char *host,
       u_long prog,
       u_long vers,
       char *proto);
```

Hostname of server
Program number
Version number
Can be "tcp" or "udp"



Calling Client Stubs

- Remember:
  - Return value is a pointer to what you expect.
  - Argument is passed as a pointer.
  - If you are passing a *string*, you must pass a **`char**`**
- When in doubt – look at the ".h" file generated by rpcgen



Server Procedures

- Rpcgen writes most of the server.
- You need to provide the actual remote procedures.
- Look in the ".h" file for prototypes.
- Run "**`rpcgen –C –Ss`**" to generate (empty) remote procedures!

Server Function Names

- Old Style (includes AIX): Remote procedure FOO, version 1 is named **`foo_1()`**
- New Style (includes Sun,BSD,Linux): Remote procedure FOO, version 1 is named **`foo_1_svc()`**

Running rpcgen

- Command line options vary from one OS to another.
- Sun/BSD/Linux – you need to use "-C" to get ANSI C code!
- Rpcgen can help write the files you need to write:
  - To generate sample server code: "-Ss"
  - To generate sample client code: "-Sc"

Other porting issues

- Shared header file generated by rpcgen may have: **`#include <rpc/rpc.h>`**
- Or Not!

RPC without rpcgen

- Can do asynchronous RPC
  - Callbacks
  - Single process is both client and server.
- Write your own dispatcher (and provide concurrency)
- Can establish control over many network parameters: protocols, timeouts, resends, etc.

`rpcinfo`

`rpcinfo –p host` prints a list of all registered programs on host.

`u` : UDP
`t` : TCP

`rpcinfo –[ut] host program#` makes a call to procedure #0 of the specified RPC program (RPC ping).

Sample Code

- simple – integer add and subtract
- ulookup – look up username and uid.
- varray – variable length array example.
- linkedlist – arg is linked list.
- rpctalk – chat program
  - doesn't work anymore :(

Example simp

- Standalone program simp.c
  - Takes 2 integers from command line and prints out the sum and difference.
  - Functions:

```
   int add( int x, int y );
int subtract( int x, int y );
```

Splitting simp.c

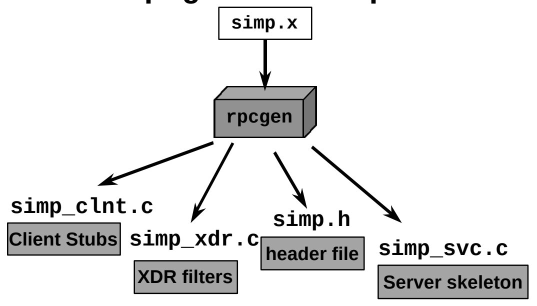
- Move the functions **`add()`** and **`subtract()`** to the server.
- Change simp.c to be an RPC client
  - Calls stubs **`add_1()`** , **`subtract_1()`**
- Create server that serves up 2 remote procedures
  - **`add_1_svc()`** and **`subtract_1_svc()`**

Protocol Definition: **`simp.x`**

```
struct operands {
        int x;
        int y;
};

program SIMP_PROG {
   version SIMP_VERSION {
     int ADD(operands) = 1;
     int SUB(operands) = 2;
   } = VERSION_NUMBER;
} = 555555555;
```

`rpcgen –C simp.x`



`xdr_operands` XDR filter

```
bool_t xdr_operands( XDR *xdrs,
               operands *objp){

  if (!xdr_int(xdrs, &objp->x))
         return (FALSE);
  if (!xdr_int(xdrs, &objp->y))
         return (FALSE);
   return (TRUE);
}
```

`simpclient.c`

- This was the main program – is now the client.
- Reads 2 ints from the command line.
- Creates a RPC handle.
- Calls the remote add and subtract procedures.
- Prints the results.

`simpservice.c`

- The server main is in **`simp_svc.c`**.
- **`simpservice.c`** is what we write – it holds the add and subtract procedures that **`simp_svc`** will call when it gets RPC requests.
- The only thing you need to do is to match the name/parameters that **`simp_svc`** expects (check **`simp.h`**!).

Userlookup program

- Provide access to passwd database via remote procedures:
  - **`getpwnam`** **`BYNAME`**
  - **`getpwuid`** **`BYNUM`**

Unix library functions **Remote Procedures**

`userlookup.x`

```
typedef string username<10>;

program ULKUP_PROG {
   version ULKUP_VERSION {
     int byname(username) = 1;
     username bynum(int) = 2;
   } = 1;
} = 555555556;
```

Problem with userlookup

- It's hard to tell if there are errors:
  - What if there is no user with the name passed to **`byname()`** ?
  - What if the username passed is not a valid username?

Better **`userlookup.h`**

```
%#define NOTFOUND 0
%#define FOUND 1

typedef string username<10>;

struct uname_retval {
    int found;
    username name;
};
```

Better **`userlookup.h`** (cont.)

```
struct uid_retval {
    int found;
    int uid;
};

program ULKUP_PROG {
   version ULKUP_VERSION {
     uid_retval byname(username) = 1;
     uname_retval bynum(int) = 2;
   } = 1;
} = 555555556;
```

Varray example

- Variable length array (determined at run time).
- Remote procedure that returns the sum of the elements in an array of int.

varray.x

```
typedef int iarray<>;

program VADD_PROG {
   version VADD_VERSION {
     int VADD(iarray) = 1;
   } = 1;
} = 555575555;
```

iarray

```
typedef int iarray<>;
```

rpcgen

```
typedef struct {
        u_int iarray_len;
        int *iarray_val;
} iarray;
```

vadd_1_svc()

```
int * vadd_1_svc(iarray *argp,
                 struct svc_req *rqstp) {
        static int  result;
        int i;

        result=0;
        for (i=0;i<argp->iarray_len;i++)
          result += argp->iarray_val[i];

        return (&result);
}
```

linkedlist

- Linked list of int.
- Remote procedure computes sum of the integers in the list.

ll.x

```
struct foo {
        int x;
        foo *next;
};

program LL_PROG {
   version LL_VERSION {
     int SUM(foo) = 1;
   } = VERSION_NUMBER;
} = 555553555;
```