Remote Procedure Call

Outline
- Concept of RPC
- SunRPC

RPC
- There exists a way for processes to represent a task: procedure/function.

Stubs
- Stub is a function with the same interface as func(); it converts the function call into a network response and a network response into a function return.
- Skeleton converts requests into function calls and function returns into network replies.
- RPC System: used to generate both stub and skeleton (automatically).

SunRPC
- Widely used Remote Procedure Call system:
  - Sun Microsystems, implemented in most Unix systems
  - NFS distributed file system is based on Sun RPC
- Designed to call remote C procedures.
- Platform-independent.
SunRPC Header Format

- XID (transaction id)
- Server does not remember last XID it serviced
- Problem if client retransmits request while reply is in transit

RPC

- One computer can be server for multiple procedures:
  - server may host several programs (identified by program number)
  - each program may have several subsequent versions (version number)
  - each version may contain one or more procedures (procedure number)
- Program numbers are 32-bit hexadecimal values (0x21212100)
  - As user, you can choose any program number between 0x20000000 and 0x3FFFFFFF, but they have to be unique (not several programs with same number on same machine)
- Version and procedure numbers are integers (1,2,...)

Sun RPC Program Numbers

<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
<th>VALUES ASSIGNED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0xFFFFFFF</td>
<td>Sun Microsystems, Inc.</td>
</tr>
<tr>
<td>0x20000000</td>
<td>0xFFFFFFF</td>
<td>System manager at a user’s site</td>
</tr>
<tr>
<td>0x40000000</td>
<td>0xFFFFFFF</td>
<td>Transient (temporary)</td>
</tr>
<tr>
<td>0x60000000</td>
<td>0xFFFFFFF</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x80000000</td>
<td>0xFFFFFFF</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xA0000000</td>
<td>0xFFFFFFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Writing an RPC Program

You write these files
rpcgen generates these files
You compile every C file

Example

- Step 1: Write a specification file (add.x)
  ```c
  struct add_in /* Arguments of procedure */
  long arg1;
  long arg2;
  
  typedef long add_out /* Return value */
  
  program ADD_PROG {
    version ADD_VERS {
      add_out ADD_PROC(add_in) = 1 /* Procedure# = 1 */
      add_out = 1 /* Version# = 1 */
    } = 1; /* Program# = 0x3434000 */
  } = 0x3434000; /* Program# = 0x3434000 */
  ```
- Step 2: Compile the C files
### Example

- Contains specifications of:
  - `'add_in'` (arguments)
  - typedef `'add_out'` (return values)
  - program `'ADD_PROG'` (0x34340000)
  - 1 version `'ADD_VERS'` (1)
  - 1 procedure `'ADD_PROC'` (1)

Your procedures can only take ONE input argument and return ONE output return value (use structures for more):
- more readable (structure entries should have meaningful names)

### Example

- Step 2: generate stubs
  - `rpcgen add.x`
- `add.h` contains declarations:
  - `#define ADD_PROG 0x34340000 /* Program nb*/`
  - `#define ADD_VERS 1 /* Version nb*/`
  - `#define ADD_PROC 1 /* Procedure nb*/`
  - `add_out * add_proc_1 (add_in *, CLIENT *)`
  - `add_out * add_proc_1_svc (add_in *, struct svc_req *)`

  - `add_proc_1` is the stub (called by client)
  - `add_proc_1_svc` is the actual procedure that you will write and run at the server

### Example

- Step 3: write server procedure: `serverproc.c`

  ```c
  #include "add.h"
  
  add_out *add_proc_1_svc(add_in *, struct svc_req *,eqtyp) {
    static add_out out;
    out = in->arg1 + in->arg2;
    return(&out);
  }
  ```

  - `rqstp` contains some information about the requester (IP address, etc.)

### Example

- Step 4: compile the server
  - Need to compile together your procedure, the (generated) server program, the (generated) marshall/unmarshall procedures and the `nsl` library (contains the RPC runtime).

  ```bash
  gcc -c serverproc.c
  gcc -c add_svc.c
  gcc -c add_xdr.c
  gcc -o server serverproc.o add_svc.o add_xdr.o -lnsl
  ```

- To start the server:
  ```bash
  ./server
  ```

### Example

- Step 5: write a client program `client.c`

  ```c
  #include "add.h"
  
  if (argc != 4) {
    printf("Usage: ...
    return 1;
  }
  
  cl = clnt_create(argv[1], ADD_PROG, ADD_VERS, "tcp"); 
  if (!cl) return(1);
  
  in.x.arg1 = atol(argv[2]);
  in.x.arg2 = atol(argv[3]);
  out = add_proc_1_svc(&in, cl);
  if (out == NULL) {
    printf("Error: %s
    clnt_sperror(cl, argv[1]));
  } else {
    printf("We received the result: %ld
    out->arg1);}
  clnt_destroy(cl);
  return 0;
  ```

- The main() returns the result of the call to `add_proc_1_svc`.
Example

- Create a client structure with clnt_create
  ```c
  #include <rpc/rpc.h>
  CLIENT *clnt_create(char *host, u_long prog, u_long vers, char *proto);
  ```
  - `host`: name of server machine
  - `prog, vers`: program/version number
  - `proto`: transport protocol ("tcp" or "udp")

- You can call add_proc_1 to send the RPC.

- When finished, destroy client structure (client structure can be used multiple times without being destroyed and recreated).

Example

- **Step 6:** compile the client
  ```bash
  $ gcc -c client.c
  $ gcc -c add_clnt.c
  $ gcc -c add_xdr.c
  $ gcc -o client client.o add_clnt.o add_xdr.o -lnsl
  ```

Example

- **Step 7:** try it out
  - start your server: ./server
  - send a request: ./client machine.cse.nd.edu 8 34
    We received the result: 42

Mutual Exclusion

- Sun RPC: at most one remote procedure in a remote program can be invoked at a given time.
- Automatic mutual exclusion among procedures within a given remote program.
- No synchronization needed.
- Some versions of rpcgen allow one to generate server code which implement one-thread-per-client (Solaris does, Linux doesn’t).

Inside SunRPC

![Inside SunRPC Diagram]

Low-Level RPC: Port Mapper

- Did you notice that we did not specify a port number for our "add" server?
  - Could be done with well-known port number for RPCs
  - But then we could not run multiple servers in the same machine simultaneously
- RPC: port mapper
  - server running on port 111
  - when your server starts, it does not bind its socket to a specific port, instead it "registers" whatever port number it has been given by the system to the local port mapper
  - when you create a client with clnt_create, you automatically contact the remote port mapper to ask for the port number of the server you want to contact
Port Mapper

You can request the port mapper by hand:

```
rpcinfo -p wizard.cse.nd.edu
```

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Proto</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>tcp</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>udp</td>
<td>32877</td>
</tr>
</tbody>
</table>

S1: Create an unbound socket (assigned port=11293)
S2: Register: program 0x3434000 version 1 running on port 11293
S3: Portmap requested: on which port is program 0x3434000 version 1?
S4: Portmap response: port 11293
S5: Run procedure 1 with parameter: "0x18A6B332F0"
S6: Result is "0x6464"

Portmapper program 56443998 version 1 running on port 11293