Lecture 12

RPC
February 9+11, 2005

RPC
- Remote Procedure Call
- Follows application-oriented approach (emphasizes problem over communication)
- Design program and then divide it.

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

Sun RPC
- 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.

Writing an RPC Program
- You write these files
- rpcgen generates these files
- You compile every C file
- You obtain a client & server

Sun RPC
- Client
  - main
  - stub
  - stub
- Server
  - Request
  - Reply
  - main
  - stub
  - stub

Writing an RPC Program
- Client
  - main
  - stub
  - stub
- Server
  - Request
  - Reply
  - main
  - stub
  - stub

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

Sun RPC
- 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.

Writing an RPC Program
- You write these files
- rpcgen generates these files
- You compile every C file
- You obtain a client & server

RPC
- Remote Procedure Call
- Follows application-oriented approach (emphasizes problem over communication)
- Design program and then divide it.
Terminology

- One computer can be server for multiple procedures:
  - Each computer 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>NAME</th>
<th>NUMBER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>portmap</td>
<td>10000</td>
<td>port mapper</td>
</tr>
<tr>
<td>rstatd</td>
<td>100001</td>
<td>rstat, nup, perfmeter</td>
</tr>
<tr>
<td>rusersd</td>
<td>100002</td>
<td>remote users</td>
</tr>
<tr>
<td>nfs</td>
<td>100003</td>
<td>network file system</td>
</tr>
<tr>
<td>ypserv</td>
<td>100004</td>
<td>yp (NIS)</td>
</tr>
<tr>
<td>mountd</td>
<td>100005</td>
<td>mount, showmount</td>
</tr>
<tr>
<td>dbxd</td>
<td>100006</td>
<td>DMXprog (debugger)</td>
</tr>
<tr>
<td>ypbind</td>
<td>100007</td>
<td>NIS binder</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Version Number

- Allows to change details of a program without obtaining a new program number.
- You can test a new version of a program while the old one still operates.

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 */
    } = 1; /* Version# = 1 */
  } = 0x3434000; /* Program# = 0x3434000 */
  ```

Example

- **Contains specifications of**:
  - `add_in` (arguments)
  - `add_out` (return values)
  - program `ADD_PROG` (0x3434000)
  - 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 *);
    ```

- **Example**
  - `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
  - `add_clnt.c`: implementation of `add_proc_1`
  - `add_svc.c`: contains program which calls your procedure `add_proc_1_svc` when request received
  - `add_xdr.c`: marshall/unmarshal routines

Example

- **Step 3**: write server procedure: `serverproc.c`
  ```c
  #include "add.h"
  add_out *add_proc_1_svc(add_in *in, struct svc_req *rqstp) {
    static add_out *out;
    out = in->arg1 + in->arg2;
    return(out);
  }
  ```

- **Example**
  - `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/unmarshal procedures and the nsl library (contains the RPC runtime).
  ```
  $ 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: `./server`

Example

- **Step 5**: write a client program `client.c`
  ```c
  #include <rpc/rpc.h>
  CLIENT *clnt_create(char *host, u_long prog, u_long vers, char *proto);
  ```

- **Example**
  - 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
  
  ```
  $ 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 Sun RPC

![Sun RPC diagram](image)

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

Low-Level RPC: Port Mapper

![Port Mapper diagram](image)

You can request the port mapper by hand:

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Protocol</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>1</td>
<td>tcp</td>
<td>111</td>
</tr>
<tr>
<td>portmapper</td>
<td>1</td>
<td>udp</td>
<td>111</td>
</tr>
<tr>
<td>66443998</td>
<td>1</td>
<td>udp</td>
<td>22877</td>
</tr>
<tr>
<td>66443938</td>
<td>1</td>
<td>tcp</td>
<td>54211</td>
</tr>
</tbody>
</table>
Low-Level RPC: Protocol
- RPC runtime does not know anything about procedure names or prototypes.
- It manages requests containing:
  - host name
  - program number
  - version number
  - procedure number
  - buffer with arguments
- Its response contains:
  - buffer with return values
- RPC runtime does not know what procedures are doing or which kinds of arguments they take
  - it only ships buffers

Low-Level RPC: Transport Issues
- RPC invocations can be with either TCP or UDP:
  - UDP is lighter/faster
  - what if message is lost?
    - after timeout client system will resend (autom.)
  - what if reply message got lost?
    - server has been invoked twice
    - RPC uses 'at-least-once' semantics
  - if you use UDP, your server must be idempotent
  - otherwise use TCP

Low-Level RPC: Transport Issues
- RPC has a simple retransmission mechanism:
1. send
2. start timer
3. wait for response
4. if timer expires: if retry limit reached stop, otherwise go back to step 1

Message Format
- Request:
  - IP header
  - UDP/TCP header
  - transaction ID (XID) 4
  - call ID (0) 4
  - RPC version (2) 4
  - program number 4
  - version number 4
  - procedure number 4
  - credentials \(8^{+}\)\(<=400\)
  - verifier \(<=408\) (secure RPC)
  - parameters

Message Format
- Response:
  - XID of request 4
  - reply (1) 4
  - status (0=accepted) 4
  - verifier \(<=408\)
  - accept status (0=accepted) 4
  - procedure results

Intermediate-Level RPC: (Un)Marshalling Parameters
- Parameters must be converted from variables into buffers (and vice-versa):
  - lowest level only knows about buffers (containing parameters)
  - user programs only know about parameters (C variables)
  - the XDR layer (external data representations) is in charge of conversion
- XDR buffer representations must be architecture-independent
  - e.g., to avoid problems when little-endian client sends requests to big-endian server...
Intermediate-Level RPC: (Un)Marshalling Parameters

- XDR library contains basic encoding/decoding functions
- Example: to (un)marshall an integer
  
  ```c
  #include <rpc/xdr.h>
  bool_t xdr_int(XDR *xdrs, int *ip);
  ```
  
  - xdrs: contains file descriptor of the socket where to read/write the buffers
  - also contains a flag to indicate whether xdr_int must encode or decode the buffer
  - Encode: convert *ip into a message buffer and write it to the socket
  - Decode: read a message buffer from the socket and convert it to *ip

High-Level RPC: Client Stub

- Client stub is in charge of providing a simple interface to client programs:
  - calls XDR functions to marshall the parameters and unmarshall the return values
  - ships them to the low-level RPC layers to be transferred to the server

  ```c
  add_out *add_proc_1(add_in *argp, CLIENT *clnt) {
    static add_out clnt_res;
    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call(clnt, ADD_PROC, (xdrproc_t)xdr_add_in, (caddr_t)argp,
                  (xdrproc_t)xdr_add_out, (caddr_t)&clnt_res, TIMEOUT) != RPC_SUCCESS) {
      return (NULL);
    }
    return (&clnt_res);
  }
  ```

High-Level RPC: Server Stub

- Server stub:
  - defines a function to reply to any request directed at it (same prog/vers number, any proc number)
  - checks the procedure number
  - calls appropriate function to unmarshall parameters
  - calls user procedure
  - marshalls the return values
  - creates UDP/TCP server sockets and registers them to port mapper
  - registers function (above) to RPC system:
    - program/version numbers
    - socket descriptors
    - procedure to serve incoming requests
  - waits for RPC requests

Summary

- RPC relates client-server communication to conventional procedure calls.
- Each server implements one or more procedures, each procedure accepts parameters and returns one or more results.
- De facto standard: Sun RPC
- Components: stubs, skeletons, RPC runtime, port mapper, rpcgen
- RPC uses XDR (external data representation) for communication.