Remote Procedure Call (RPC)

- Paradigms in building distributed applications
- The RPC model
- Primitives
- Issues
- Case study: Sun RPC
- Reading: Coulouris, Chapter 5

Building Distributed Programs: Two Paradigms

Paradigms:
- Communication-Oriented Design
  - Start with communication protocol
  - Design message format and syntax
  - Design client and server components by specifying how they react to incoming messages
- Application-Oriented Design
  - Start with application
  - Design, build, test conventional implementation
  - Partition program

Problems:
- Protocol-design problems
- Application components as finite-state machines?
- Focus on communication instead of application
- Concurrency
Model of Execution for RPCs

- Procedure-call structure of a program

Machine 1
- main
- proc A
- proc B

Machine 2
- proc A

Machine 3
- proc B

Model of execution with remote procedure call

- call remote proc A
- call remote proc B
- respond to caller
- respond to caller

RPC Properties

- Uniform call structure
- Type checking
- Full parameter functionality
- Distributed binding
- Recovery of orphan computations
RPC Primitives

- Invocation at caller side
  ```
  call service (value_args; result_args);
  ```

- Definition at server side
  - declaration
    ```
    remote procedure service (in value_pars;
    out result_pars);
    begin body end;
    ```
  - rendezvous statement
    ```
    accept service (in value_pars;
    out result_pars) -> body;
    ```

Structure of an RPC Call

1. Client
2. Client stubs
3. RPC library
4. Server stubs
5. Server
6. Remote procedure
7. Accept statement
8. Call
9. Service
10. Return result
RPCs: Issues

- Parameter passing
  - value parameters
  - reference parameters?
- Marshalling
  - simple data types
  - complex data structures
- Exception handling
  - language dependent
  - need to deal with asynchronous events

Locating Servers

- Broadcast requests
  - broadcast call and process incoming replies
- Name servers
  - server registers with name server

- Combination: publish/subscribe
Communication Protocols for RPC

- Reliable protocols: e.g. TCP
- Unreliable datagram protocols: e.g. UDP
- Specifically designed protocols: Example

### Simple Call

- (id, request)
- (id, reply, ack)
- (id, request)
- (id, reply, ack)

Client times out and retransmits request. Three cases:
- request lost
- server still executing
- ack lost

### Complicated Call

- long gaps between requests
- acknowledge each message transmission separately or
- periodically send “I-am-alive” message and use simple-call scheme.
- long messages (don’t fit into packet)
- segment message
- segment-relative seq #’s
- retransmission scheme for segments

RPC in Heterogeneous Environments

- Compile-time support
- Binding protocol
- Transport protocol
- Control protocol
- Data representation
Case Study: SUN RPC

- Defines format for messages, arguments, and results.
- Uses UDP or TCP.
- Uses XDR (eXternal Data Representation) to represent procedure arguments and header data.
- Compiler system to automatically generate distributed programs.
- Remote execution environment: remote program.

![Diagram of remote program and shared data]

- Mutually exclusive execution of procedure in remote program.

Identifying Remote Programs and Procedures

- Conceptually, each procedure on a computer is identified by pair:

  \((prog, proc)\)

  - \(prog\): 32-bit integer identifying remote program
  - \(proc\): integer identifying procedure

- Set of program numbers partitioned into 8 sets.

<table>
<thead>
<tr>
<th>Program Range</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0xffffffff</td>
<td>assigned by SUN</td>
</tr>
<tr>
<td>0x20000000 - 0x3fffffff</td>
<td>assigned by local system manager</td>
</tr>
<tr>
<td>0x40000000 - 0x5fffffff</td>
<td>temporary</td>
</tr>
<tr>
<td>0x60000000 - 0xffffffff</td>
<td>reserved</td>
</tr>
</tbody>
</table>

- Multiple remote program versions can be identified:

  \((prog, version, proc)\)
## Example RPC Program Numbers

<table>
<thead>
<tr>
<th>name</th>
<th>assigned no</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>portmap</td>
<td>100000</td>
<td>port mapper</td>
</tr>
<tr>
<td>rstatd</td>
<td>100001</td>
<td>rstat, rup, 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>DBXprog (debug)</td>
</tr>
<tr>
<td>yppbind</td>
<td>100007</td>
<td>NIS binder</td>
</tr>
<tr>
<td>walld</td>
<td>100008</td>
<td>rwall, shutdown</td>
</tr>
<tr>
<td>yppasswdd</td>
<td>100009</td>
<td>yppassword</td>
</tr>
</tbody>
</table>

## Communication Semantics

- TCP or UDP?
- Sun RPC semantics defined as function of underlying transport protocol.
  - RPC on UDP: calls can be lost or duplicated.
- *at-least-once* semantics if caller receives reply.
- *zero-or-more* semantics if caller does not receive reply.
- Programming with *zero-or-more* semantics: *idempotent* procedure calls.
- Sun RPC retransmission mechanism:
  - non-adaptive timeouts
  - fixed number of retransmissions
Remote Programs and Protocol Ports

- Dynamic port mapping: RPC port mapper

Sun RPC Message Format: XDR Specification

```c
enum msg_type { /* RPC message type constants */
    CALL = 0;
    REPLY = 1;
};

struct rpc_msg { /* format of a RPC message */
    unsigned int mesgid; /* used to match reply to call */
    union switch (msg_type mesgt) {
        case CALL : call_body cbody;
        case REPLY: reply_body rbody;
    } body;
};

struct call_body { /* format of RPC CALL */
    u_int rpcvers; /* which version of RPC? */
    u_int rprog; /* remote program number */
    u_int rprogvers; /* version number of remote prog */
    u_int rproc; /* number of remote procedure */
    opaque_auth cred; /* credentials for called auth. */
    opaque_auth verf; /* authentication verifier */
    /* ARGS */
};
```
Message Dispatch for Remote Programs

Creating Distributed Applications with Sun RPC
Example: Remote Dictionary Using `rpcgen`

- Procedure call structure:

Procedures should execute on the same machines as their resources are located.
**Specification for rpcgen**

```
/* rdict.x */
/* RPC declarations for dictionary program */
const MAXWORD = 50;
const DICTSIZ = 100;
struct example { /* unused; rpcgen would */
    int exfield1; /* generate XDR routines */
    char exfield2; /* to convert this structure. */
};

/* RDICTPROG: remote program that provides
  insert, delete, and lookup */
program RDICTPROG { /* name (not used) */
    version RDICTVERS { /* version declarat. */
        int INITW(void) = 1; /* first procedure */
        int INSERTW(string) = 2; /* second proc.... */
        int DELETENW(string) = 3;
        int LOOKUP(string) = 4;
    } = 1; /* version definit. */
} = 0x30090949; /* program no */
/* (must be unique) */
```

**Program Generation**

```
rpcgen rdict.x
```

- rdict.h
  - constants, datatypes
  - definitions for remote procedures

- rdict_xdr.c
  - XDR conversion routines

- rdict_clnt.c
  - client code: client-side communication stub.

- rdict_svc.c
  - server code: server-side communication stub.

- rdict_sif.c
- rdict2.c
- rdict (client)
- rdictd
- cc
- rdict_cif.c
- rdict1.c
- rdict_clnt.c
- rdict_xdr.c
- rdict_svc.c
- rdict_sif.c
- rdict2.c