The RPC abstraction

- Procedure calls well-understood mechanism
  - Transfer control and data on single computer

- Goal: Make distributed programming look same
  - Code libraries provide APIs to access functionality
  - Have servers export interfaces accessible through local APIs

- Implement RPC through request-response protocol
  - Procedure call generates network request to server
  - Server return generates response
RPC Failure

- More failure modes than simple procedure calls
  - Machine failures
  - Communication failures
- RPCs can return “failure” instead of results
- What are possible outcomes of failure?
  - Procedure did not execute
  - Procedure executed once
  - Procedure executed multiple times
  - Procedure partially executed
- Generally desired semantics: at most once
Implementing at most once semantics

• **Danger: Request message lost**
  - Client must retransmit requests when it gets no reply

• **Danger: Reply message may be lost**
  - Client may retransmit previously executed request
  - Okay if operations are idempotent, but many are not (e.g., process order, charge customer, …)
  - Server must keep “replay cache” to reply to already executed requests

• **Danger: Server takes too long to execute procedure**
  - Client will retransmit request already in progress
  - Server must recognize duplicate—can reply “in progress”
Server crashes

- **Danger: Server crashes and reply lost**
  - Can make replay cache persistent—slow
  - Can hope reboot takes long enough for all clients to fail

- **Danger: Server crashes during execution**
  - Can log enough to restart partial execution—slow and hard
  - Can hope reboot takes long enough for all clients to fail
Parameter passing

- Different data representations
  - Big/little endian
  - Size of data types

- No shared memory
  - No global variables
  - How to pass pointers
  - How to garbage collect distributed objects

- How to pass unions
Interface Definition Languages

• Idea: Specify RPC call and return types in IDL

• Compile interface description with IDL compiler.
  Output:
  - Native language types (e.g., C/Java/C++ structs/classes)
  - Code to **marshal** (serialize) native types into byte streams
  - **Stub** routines on client to forward requests to server

• **Stub routines handle communication details**
  - Helps maintain RPC transparency, but
  - Still had to bind client to a particular server
  - Still need to worry about failures
Intro to SUN RPC

• Simple, no-frills, widely-used RPC standard
  - Does not emulate pointer passing or distributed objects
  - Programs and procedures simply referenced by numbers
  - Client must know server—no automatic location
  - Portmap service maps program #s to TCP/UDP port #s

• IDL: XDR – eXternal Data Representation
  - Compilers for multiple languages (C, java, C++)
Transport layer

- Transport layer transmits delimited RPC messages
  - In theory, RPC is transport-independent
  - In practice, RPC library must know certain properties (e.g., Is transport connected? Is it reliable?)

- UDP transport: unconnected, unreliable
  - Sends one UDP packet for each RPC request/response
  - Each message has its own destination address
  - Server needs replay cache

- TCP transport (simplified): connected, reliable
  - Each message in stream prefixed by length
  - RPC library does not retransmit or keep replay cache
Sun XDR

- "External Data Representation"
  - Describes argument and result types:
    
    ```c
    struct message {
        int opcode;
        opaque cookie[8];
        string name<255>;
    };
    ```
  - Types can be passed across the network

- **Libasync rpcc compiles to C++**
  - Converts messages to native data structures
  - Generates marshaling routines (struct ↔ byte stream)
  - Generates info for stub routines
Basic data types

- **int var** – 32-bit signed integer
  - wire rep: big endian (0x11223344 → 0x11, 0x22, 0x33, 0x44)
  - rpcc rep: int32_t var

- **hyper var** – 64-bit signed integer
  - wire rep: big endian
  - rpcc rep: int64_t var

- **unsigned int var, unsigned hyper var**
  - wire rep: same as signed
  - rpcc rep: u_int32_t var, u_int64_t var
More basic types

- **void** – No data
  - wire rep: 0 bytes of data

- **enum** {name = constant,...} – enumeration
  - wire rep: Same as int
  - rpc rep: enum

- **bool var** – boolean
  - both reps: As if enum bool {FALSE = 0, TRUE = 1} var
Opaque data

- **opaque var[n]** – *n* bytes of opaque data
  - wire rep: *n* bytes of data, 0-padded to multiple of 4
    - opaque v[5] → v[0], v[1], v[2], v[3], v[4], 0, 0, 0
  - rpcc rep: rpc_opaque<n> var
    - var[i]: char & – *i*-th byte
    - var.size (): size_t – number of bytes (i.e. *n*)
    - var.base (): char * – address of first byte
    - var.lim (): char * – one past last
Variable length opaque data

- **opaque var<n>** – 0–n bytes of opaque data
  - wire rep: 4-byte data size in big endian format, followed by n bytes of data, 0-padded to multiple of 4
  - rpc rep: `rpc_bytes<n> var`
  - `var.setsize(size_t n)`: set size to n (destructive)
  - `var[i]`: `char &` – i-th byte
  - `var.size()`: `size_t` – number of bytes
  - `var.base()`: `char *` – address of first byte
  - `var.lim()`: `char *` – one past last

- **opaque var<>** – arbitrary length opaque data
  - wire rep: same
  - rpc rep: `rpc_bytes<RPC_INFINITY> var`
Strings

• **string var\(n\)** – **string of up to** \(n\) **bytes**
  - wire rep: just like opaque var\(n\)
  - rpcc rep: rpc_str\(n\) behaves like str, except cannot be NULL, cannot be longer than \(n\) bytes

• **string var<>** – **arbitrary length string**
  - wire rep: same as string var\(n\)
  - rpcc rep: same as string var\(RPC\_INFINITY\)

• **Note: Strings cannot contain 0-valued bytes**
  - Should be allowed by RFC
  - Because of C string implementations, does not work
  - rpcc preserves “broken” semantics of C applications
Arrays

- **obj_t var[n]** – *Array of n objects*
  - wire rep: n wire reps of obj_t in a row
  - rpcc rep: array<obj_t, n> var; as for opaque:
    var[i], var.size (), var.base (), var.lim ()

- **obj_t var<n>** – *0–n objects*
  - wire rep: array size in big endian, followed by that many wire reps of obj_t
  - rpcc rep: rpc_vec<obj_t, n> var; var.setsize (n),
    var[i], var.size (), var.base (), var.lim ()
Pointers

- obj_t *var – “optional” obj_t
  - wire rep: same as obj_t var<1>: Either just 0, or 1 followed by wire rep of obj_t
  - rpcc rep: rpc_ptr<obj_t> var
    - var.alloc () – makes var behave like obj_t *
    - var.clear () – makes var behave like NULL
    - var = var2 – Makes a copy of *var2 if non-NLL

- Pointers allow linked lists:

  struct entry {
    filename name;
    entry *nextentry;
  };

- Not to be confused with network object pointers!
Structures

struct type {
    type_A fieldA;
    type_B fieldB;
    ...
};

- wire rep: wire representation of each field in order
- rpcc rep: structure as defined
Discriminated unions

```
union type switch (simple_type which) {
    case value_A:
        type_A varA;
    ...
    default:
        void;
};
```

- simple_type must be [unsigned] int, bool, or enum
- Wire representation: wire rep of which, followed by wire rep of case selected by which.
Discriminated unions: rpcc representation

```c
struct type {
    simple_type which;
    union {
        union_entry<type_A> varA;
        ...
    };
};
```

- void type::set_which (simple_type newwhich)
  sets the value of the discriminant

- varA behaves like type_A * if which == value_A

- Otherwise, accessing varA causes core dump
  (when using dmalloc)
Example: fetch and add server

```c
struct fadd_arg {
    string var<>;
    int inc;
};

union fadd_res switch (int error) {
    case 0:
        int sum;
    default:
        void;
};
```
RPC program definition

program FADD_PROG {
    version FADD_VERS {
        void FADDPROC_NULL (void) = 0;
        fadd_res FADDPROC_FADD (fadd_arg) = 1;
    } = 1;
} = 300001;
Client code

fadd_arg arg; fadd_res res;

void getres (clnt_stat err) {
    if (err) warn << "server: " << err << "\n";  // pretty-prints
    else if (res.error) warn << "error #" << res.error << "\n";
    else warn << "sum is " << *res.sum << "\n";
}

void start () {
    int fd;
    /* ... connect fd to server, fill in arg ... */
    ref<axprrt> x = axprrt_stream::alloc (fd);
    ref<aclnt> c = aclnt::alloc (x, fadd_prog_1);
    c->call (FADDPROC_FADD, &arg, &res, wrap (getres));
}
Server code

qhash<str, int> table;
void dofadd (fadd_arg *arg, fad_res *res) {
    int *valp = table[arg->var];
    if (valp) {
        res.set_error (0);
        *res->sum = *valp += arg->inc;
    } else
        res.set_error (NOTFOUND);
}

ptr<asrv> s;
void getnewclient (int fd) {
    s = asrv::alloc (axprt_stream::alloc (fd), fadd_prog_1,
        wrap (dispatch));
}
Server dispatch code

```c
void dispatch (svccb *sbp) {
    if (!sbp) { s = NULL; return; }
    switch (sbp->proc ()) {
    case FADDPROC_NULL:
        sbp->reply (NULL);
        break;
    case FADDPROC_FADD:
        fadd_res res;
        dofadd (sbp->template getarg<fadd_arg> (), &res);
        sbp->reply (&res);
        break;
    default:
        sbp->reject (PROC_UNAVAIL);
    }
}
```
NFS version 2

• **Background: ND**
  - What is ND?
  - How is NFS different from ND?
  - Why might NFS be slower than ND?

• **Some Goals of NFS**
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND

• **Did they achieve goals?**
Stateless operation

- Goal: server crash recovery
- Requests are self-contained
- Requests are idempotent
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
  - Does this really work?
Semantics

- Component-by-component lookup
- Hard/soft mount
- Credentials and authentication
- Unlinking open files
- Permissions on open files
- Cache consistency
Optimizations

• NFS server and block I/O daemons
• Client-side buffer cache (write-behind w. flush-on-close)
• XDR directly to/from mbufs
• Client-side attribute cache
• Fill-on-demand clustering, swap in small programs
• Name cache