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

● Can use “cookies” to inform clients of crashes
  - Server gives client cookie which is time of boot
  - Client includes cookie with RPC
  - After server crash, server will reject invalid cookie
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:
    
    ```
    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
  - rpcc 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
  - rpcc 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
  - rpcc 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 obj_ts**
  - 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 obj_ts**
  - 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-NULL

- Pointers allow linked lists:

  ```c
  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)
RPC message format

```c
enum msg_type { CALL = 0, REPLY = 1 };  
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};
```

- **32-bit XID identifies each RPC**
  - Chosen by client, returned by server
  - Server may base replay cache on XID
RPC call format

```c
struct call_body {
    unsigned int rpcvers; /* must always be 2 */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* argument structure goes here */
};
```

- Every call has a 32-bit program & version number
  - E.g., NFS is program 100003, versions 2 & 3 in use
  - Can implement multiple servers on same port

- Opaque auth is hook for authentication & security
  - Credentials – who you are; Verifier – proof.
RPC reply format

enum reply_stat { MSG_ACCEPTED = 0, MSG_DENIED = 1 }; union reply_body switch (reply_stat stat) {
  case MSG_ACCEPTED:
    accepted_reply areply;
  case MSG_DENIED:
    rejected_reply rreply;
}

- Most calls generate “accepted replies”
  - Includes many error conditions, too

- Authentication failures produce “rejected replies”
Accepted calls

```c
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            /* result structure goes here */
        case PROG_MISMATCH:
            struct { unsigned low; unsigned high; } mismatch_info;
        default:
            /* PROG/PROC_UNAVAIL, GARBAGE_ARGS, SYSTEM_ERR, ... */
            void;
    } reply_data;
};
```
Rejected calls

define reject_stat { RPC_MISMATCH = 0, AUTH_ERROR = 1 }
union rejected_reply switch (reject_stat stat) {
case RPC_MISMATCH:
    struct {
        unsigned int low;
        unsigned int high;
    } mismatch_info; /* means rpcvers != 2 */
case AUTH_ERROR: /* means rpcvers != 2 */
    auth_stat stat; /* Authentication insufficient */
};
RPC authentication

enum auth_flavor {
    AUTH_NONE = 0,
    AUTH_SYS = 1,  /* a.k.a. AUTH_UNIX */
    AUTH_SHORT = 2,
    AUTH_DES = 3
};

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};

- Opaque allows new types w/o changing RPC lib
  - E.g., SFS adds AUTH_UINT=10, containing simple integer
AUTH_UNIX credential flavors

```c
struct authsys_parms {
    unsigned int time;
    string machinename<255>;
    unsigned int uid;
    unsigned int gid;
    unsigned int gids<16>;
};
```

- Contains credentials of user on client machine
- Only useful if:
  1. Server trusts client machine, and
  2. Client and server have same UIDs/GIDs, and
  3. Network between client and server is secure
Example: fetch and add server

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;

• RPC library needs information for each call
  - prog, vers, marshaling function for arg and result

• rpcc encapsulates all needed info in a struct
  - Lower-case prog name, numeric version: fadd_prog_1
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<axprt> x = axprt_stream::alloc (fd);
  ref<aclnt> c = aclnt::alloc (x, fadd_prog_1);
  c->call (FADDPROC_FADD, &arg, &res, wrap (getres));
}
Server code

```c
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 3

- **Same general architecture as NFS 2**
  - Maybe saw in CS240

- **Specified in RFC 1813**
  - See class reference materials web page
  - Will need NFS3 spec and XDR spec (RFC 1832)
File handles

struct nfs_fh3 {
    opaque data<64>;
};

- **Server assigns an opaque file handle to each file**
  - Client obtains first file handle out-of-band (mount protocol)
  - File handle hard to guess – security enforced at mount time
  - Subsequent file handles obtained through lookups

- **File handle internally specifies file system / file**
  - Device number, i-number, generation number, …
  - Generation number changes when inode recycled
File attributes

```c
struct fatr3 {  
    specdata3 rdev;
    ftype3 type;
    uint64 fsid;
    uint32 mode;
    uint64 fileid;
    uint32 nlink;
    nfstime3 atime;
    uint32 uid;
    nfstime3 mtime;
    uint32 gid;
    nfstime3 ctime;
    uint64 size;
};
uint64 used;
```

- Most operations can optionally return `fatr3`
- Attributes used for cache-consistency
Lookup

struct diropargs3 {
  nfs_fh3 dir;
  filename3 name;
};

struct lookup3resok {
  nfs_fh3 object;
  post_op_attr obj_attributes;
  post_op_attr dir_attributes;
};

union lookup3res switch (nfsstat3 status) {
  case NFS3_OK:
    lookup3resok resok;
    break;
  default:
    post_op_attr resfail;
    break;
};

- **Maps** \( \langle \text{directory, handle} \rangle \rightarrow \text{handle} \)
  - Client walks hierarchy one file at a time
  - No symlinks or file system boundaries crossed
Create

struct create3args {
    union createhow3 switch (createmode3 mode) {
        diropargs3 where;
        createhow3 how;
    }
    sattr3 obj_attributes;
    case EXCLUSIVE:
        createverf3 verf;
    }

- UNCHECKED – succeed if file exists
- GUARDED – fail if file exists
- EXCLUSIVE – persistent record of create
Read

struct read3args {
    nfs_fh3 file;
    uint64 offset;
    uint32 count;
};

struct read3resok {
    post_op_attr file_attributes;
    uint32 count;
    bool eof;
    opaque data<>;
};

union read3res switch (nfsstat3 status) {
    case NFS3_OK:
        read3resok resok;
    default:
        post_op_attr resfail;
};

- Offset explicitly specified (not implicit in handle)
- Client can cache result
Data caching

- Client can cache blocks of data read and written

- **Consistency based on times in `fattr3`**
  - `mtime`: Time of last modification to file
  - `ctime`: Time of last change to inode
    (Changed by explicitly setting mtime, increasing size of file, changing permissions, etc.)

- **Algorithm**: If mtime or ctime changed by another client, flush cached file blocks
Write discussion

• When is it okay to lose data after a crash?
  - Local file system
  - Network file system

• NFS2 servers flush writes to disk before returning
  - Caused performance problems
CS240 Flashback: NFS2 write call

```c
struct writeargs {
    fhandle file;
    unsigned beginoffset;
    unsigned offset;
    unsigned totalcount;
    nfsdata data;
};

union attrstat {
    switch (stat status) {
        case NFS_OK:
            fattr attributes;
        default:
            void;
    }
};

attrstat NFSPROC_WRITE(writeargs) = 8;
```

- On successful write, returns new file attributes
- Can NFS2 keep cached copy of file after writing it?
Write race condition

- Suppose client overwrites 2-block file
  - Client A knows attributes of file after writes A1 & A2
  - But client B could overwrite block 1 between the A1 & A2
  - No way for client A to know this hasn’t happened
  - Must flush cache before next file read (or at least open)
NFS3 Write arguments

```c
struct write3args {
    nfs_fh3 file;
    uint64 offset;
    uint32 count;
    stable_how stable;
    opaque data<>;
};
```

```c
eenum stable_how {
    UNSTABLE = 0,
    DATA_SYNC = 1,
    FILE_SYNC = 2
};
```

- **Two goals for NFS3 write:**
  - Don’t force clients to flush cache after writes
  - Don’t equate cache consistency with crash consistency
    I.e., don’t wait for disk just so another client can see data
Write results

```c
struct write3resok {
    wcc_data file_wcc;
    uint32 count;
    stable_how committed;
    writeverf3 verf;
};
union write3res {
    struct write3resok resok;
    switch (nfsstat3 status) {
    case NFS3_OK:
        write3resok resok;
    default:
        wcc_data resfail;
    }
};

- Several fields added to achieve these goals
```
Data caching after a write

- **Write will change mtime/ctime of a file**
  - “after” will contain new times
  - Should cause cache to be flushed

- **“before” contains previous values**
  - If before matches cached values, no other client has changed file
  - Okay to update attributes without flushing data cache
Write stability

- Server write must be at least as stable as requested.
- If server returns write UNSTABLE:
  - Means permissions okay, enough free disk space, …
  - But data not on disk and might disappear (after crash)
- If DATA_SYNC, data on disk, maybe not attributes
- If FILE_SYNC, operation complete and stable
Commit operation

- **Client cannot discard any UNSTABLE write**
  - If server crashes, data will be lost

- **COMMIT RPC commits a range of a file to disk**
  - Invoked by client when client cleaning buffer cache
  - Invoked by client when user closes/flushes a file

- **How does client know if server crashed?**
  - Write and commit return `writeverf3`
  - Value changes after each server crash (may be boot time)
  - Client must resend all writes if `verf` value changes
Attribute caching

● Close-to-open consistency
  - It really sucks if writes not visible after a file close
    (Edit file, compile on another machine, get old version)
  - Nowadays, all NFS opens fetch attributes from server

● Still, lots of other need for attributes (e.g., `ls -al`)

● Attributes cached between 5 and 60 seconds
  - Files recently changed more likely to change again
  - Do weighted cache expiration based on age of file

● Drawbacks:
  - Must pay for round-trip to server on every file open
  - Can get stale info when statting a file