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

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++)

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 $\leftrightarrow$ 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_t`**s**
  - 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_t`**s**
  - 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:**

```
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

```
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

```
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

```
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

```
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);
  }
}
```

NFS3: 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

```
struct fattr3 {
  ftype3 type;
  uint32 mode;
  uint32 nlink;
  uint32 uid;
  uint32 gid;
  uint64 size;
  uint64 used;
  specdata3 rdev;
  uint64 fsid;
  uint64 fileid;
  nfstime3 atime;
  nfstime3 mtime;
  nfstime3 ctime;
};
```

- **Most operations can optionally return** `fattr3`
- **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;
default:
  post_op_attr resfail;
};
```

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

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**

NFS3 Write arguments

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

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

Write results

```
struct write3resok {
  wcc_data file_wcc;
  uint32 count;
  stable_how committed;
  writeverf3 verf;
};

union write3res switch (nfsstat3 status) {
case NFS3_OK:
  write3resok resok;
default:
  wcc_data resfail;
};
```

```
struct wcc_attr {
  uint64 size;
  nfstime3 mtime;
  nfstime3 ctime;
};
struct wcc_data {
  wcc_attr *before;
  post_op_attr after;
};
```

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

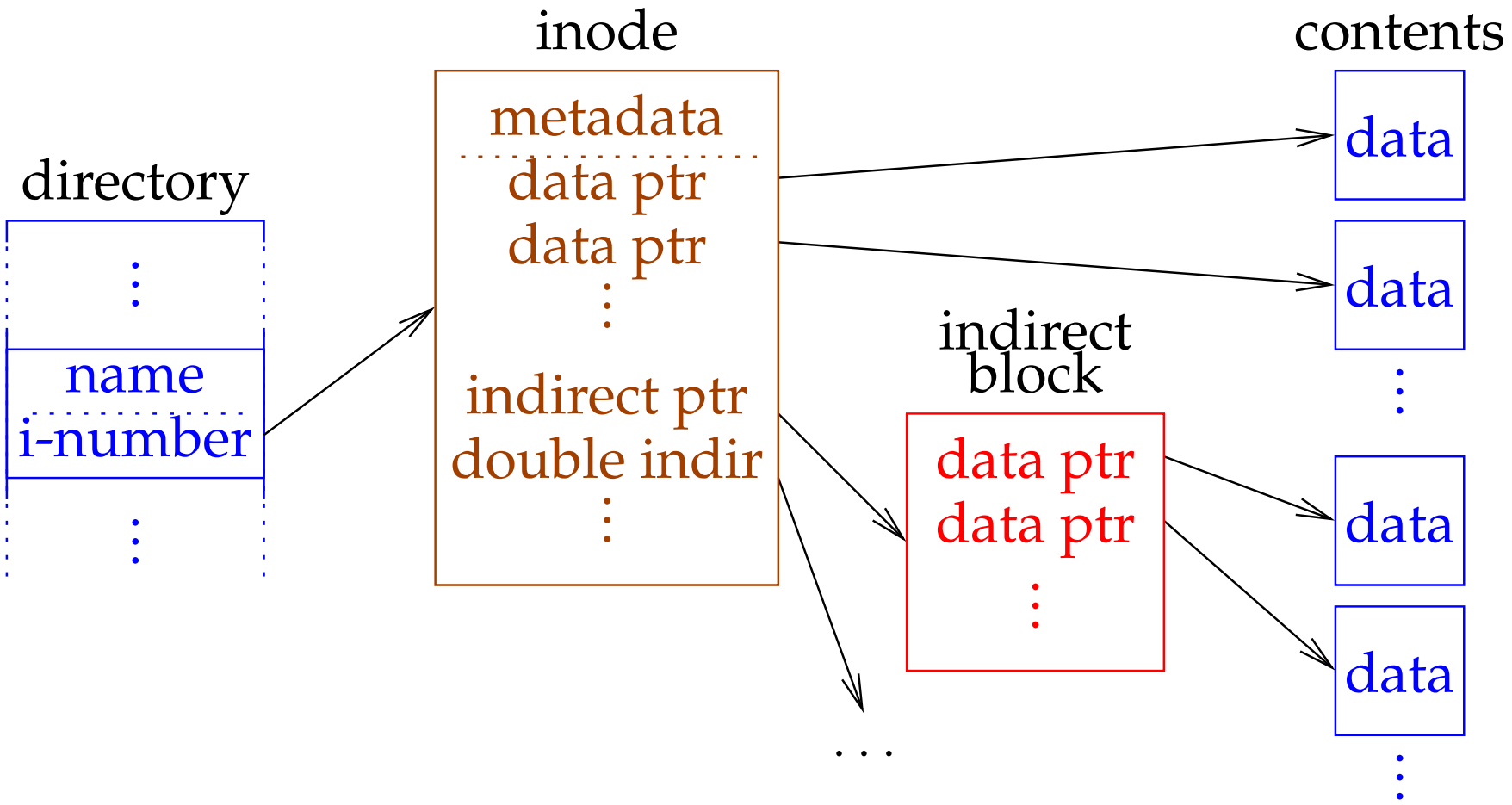
FFS: Back in the 80s…

- **Disks spin at 3,600 RPM**
  - 17 ms/Rotation (vs. 4 ms on fastest disks today)
- **Fixed # sectors/track (no zoning)**
- **Head switching free (?)**
- **Requests issued one at a time**
  - No caching in disks
  - Head must pass over sector after getting a read
  - By the time OS issues next request, too late for next sector
- **Slower CPUs, memory**
  - Noticeable cost for block allocation algorithms

Original Unix file system

- **Each FS breaks partition into three regions:**
  - Superblock (parameters of file system, free ptr)
  - Inodes – type/mode/size + ptr to data blocks
  - File and directory data blocks
- **All data blocks 512 bytes**
- **Free blocks kept in a linked list**

Inodes


inode
contents
directory
metadata
data ptr
data ptr
indirect ptr
double indir
name
i-number
indirect
block
data ptr
data ptr
data
data
data
data
. . .

Problems with original FS

- **FS never transfers more than 512 bytes/disk access**
- **After a while, allocation essentially random**
  - Requires a random seek every 512 bytes of file data
- **Inodes far from both directory data and file data**
- **Within a directory, inodes not near each other**
- **Usability problems:**
  - File names limited to 14 characters
  - No way to update file atomically & guarantee existence after crash

Fast file system

- **New block size must be at least 4K**
  - To avoid wasting space, use “fragments” for ends of files
- **Cylinder groups avoid spread inodes around disk**
- **Bitmaps replace free list**
- **FS reserves space to improve allocation**
  - Tunable parameter, default 10%
  - Only superuser can use space when over 90% full

FFS superblock

- **Contains file system parameters**
  - Disk characteristics, block size, CG info
  - Information necessary to get inode given i-number
- **Replicated once per cylinder group**
  - At shifting offsets, so as to span multiple platters
  - Contains magic to find replicas if 1st superblock dies
- **Contains non-replicated "summary info"**
  - # blocks, fragments, inodes, directories in FS
  - Flag stating if FS was cleanly unmounted

Cylinder groups

- **Groups related inodes and their data**
- **Contains a number of inodes (set when FS created)**
  - Default one inode per 2K data
- **Contains file and directory blocks**
- **Contains bookkeeping information**
  - Block map – bit map of available fragments
  - Summary info within CG – # free inodes, blocks/frags, files, directories
  - # free blocks by rotational position (8 positions)

Inode allocation

- **Allocate inodes in same CG as directory if possible**
- **New directories put in new cylinder groups**
  - Consider CGs with greater than average # free inodes
  - Chose CG with smallest # directories
- **Within CG, inodes allocated randomly (next free)**
  - Would like related inodes as close as possible
  - OK, because one CG doesn't have that many inodes

Fragment allocation

- **Allocate space when user writes beyond end of file**
- **Want last block to be a fragment if not full-size**
  - If already a fragment, may contain space for write – done
  - Else, must deallocate any existing fragment, allocate new
- **If no appropriate free fragments, break full block**
- **Problem: Slow for many small writes**
  - (Partial) soution: new `stat` struct field `st_blksize`
  - Tells applications file system block size
  - stdio library can buffer this much data

Block allocation

- **Try to optimize for sequential access**
  - If available, use rotationally close block in same cylinder
  - Otherwise, use block in same CG
  - If CG totally full, find other CG with quadratic hashing
  - Otherwise, search all CGs for some free space
- **Problem: Don't want one file filling up whole CG**
  - Otherwise other inodes will have data far away
- **Solution: Break big files over many CGs**
  - But large extents in each CGs, so sequential access doesn't require many seeks

Directories

- **Inodes like files, but with different type bits**
- **Contents considered as 512-byte *chunks***
- **Each chunk has `direct` structure(s) with:**
  - 32-bit inumber
  - 16-bit size of directory entry
  - 8-bit file type (NEW)
  - 8-bit length of file name
- **Coalesce when deleting**
  - If first `direct` in chunk deleted, set inumber = 0
- **Periodically compact directory chunks**

Updating FFS for the 90s

- **No longer want to assume rotational delay**
  - With disk caches, want data contiguously allocated
- **Solution: Cluster writes**
  - FS delays writing a block back to get more blocks
  - Accumulates blocks into 64K clusters, written at once
- **Allocation of clusters similar to fragments/blocks**
  - Summary info
  - Cluster map has one bit for each 64K if all free
- **Also read in 64K chunks when doing read ahead**

Dealing with crashes

- **Suppose all data written asynchronously**
- **Delete/truncate a file, append to other file, crash**
  - New file may reuse block from old
  - Old inode may not be updated
  - Cross-allocation!
  - Often inode with older mtime wrong, but can't be sure
- **Append to file, allocate indirect block, crash**
  - Inode points to indirect block
  - But indirect block may contain garbage

Ordering of updates

- **Must be careful about order of updates**
  - Write new inode to disk before directory entry
  - Remove directory name before deallocating inode
  - Write cleared inode to disk before updating CG free map
- **Solution: Many metadata updates syncrhonous**
  - Of course, this hurts performance
  - E.g., untar much slower than disk b/w
- **Note: Cannot update buffers on the disk queue**

Fixing corruption – fsck

- **Summary info usually bad after crash**
  - Scan to check free block map, block/inode counts
- **System may have corrupt inodes (not simple crash)**
  - Bad block numbers, cross-allocation, etc.
  - Do sanity check, clear inodes with garbage
- **Fields in inodes may be wrong**
  - Count number of directory entries to verify link count, if no entries but count $\neq 0$, move to `lost+found`
  - Make sure size and used data counts match blocks
- **Directories may be bad**
  - Holes illegal, `.` and `..` must be valid, ...
  - All directories must be reachable