Administrivia

- **Project 3 due Friday noon**
- **If you need longer, email** `cs140-staff`
- **Remember Friday section on Project 4**

Network file systems

- **What's a network file system?**
  - Looks like a file system (e.g., FFS) to applications
  - But data potentially stored on another machine
  - Reads and writes must go over the network
  - Also called distributed file systems
- **Advantages of network file systems**
  - Easy to share if files available on multiple machines
  - Often easier to administer servers than clients
  - Access way more data than fits on your local disk
  - Network + remote buffer cache faster than local disk
- **Disadvantages**
  - Network + remote disk slower than local disk
  - Network or server may fail even when client OK
  - Complexity, security issues

Naming structures

- **Location transparency – file name does not reveal the file's physical storage location**
  - File name still denotes a specific, although hidden, set of physical disk blocks
  - Some naming structure enforced by OS / file system
  - Can expose correspondence between component units and machines
- **Location independence – file name need not change when the file's physical storage location changes**
  - Better file abstraction
  - Promotes sharing the storage space itself
  - Separates the naming hierarchy form the storage-devices hierarchy

Naming schemes

- **File names include server and local path (URLs)**
  - E.g., `http://server.stanford.edu/home/dm` – unique name
  - Variation: Include cryptographically secure name for server
- **Attach remote directories to local directories (NFS)**
  - Gives appearance of a coherent directory tree
  - Only previously mounted remote directories accessible
- **Total integration of the component file systems (AFS)**
  - A single global name structure spans all the files in the system
  - If a server is unavailable, some set of directories on also becomes unavailable on all clients
- **Name by the data you want (Chord CFS, IVY)**
  - Very non-standard administrative model (impractical?)
  - Some big advantages like scalability & fault tolerance

NFS version 2 [Sandberg]

- **Background: ND (networked disk)**
  - Creates disk-like device even on diskless workstations
  - Can create a regular (e.g., FFS) file system on it
  - But no sharing—Why?
- **ND idea still used today by Linux** `nbd`
  - Useful for network booting/diskless machines, not file sharing
- **Some Goals of NFS**
  - Access same FS from multiple machines simultaneously
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND

NFS version 2 [Sandberg]

- **Background: ND (networked disk)**
  - Creates disk-like device even on diskless workstations
  - Can create a regular (e.g., FFS) file system on it
  - But no sharing—Why?
  - FFS assumes disk doesn't change under it
- **ND idea still used today by Linux** `nbd`
  - Useful for network booting/diskless machines, not file sharing
- **Some Goals of NFS**
  - Access same FS from multiple machines simultaneously
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND

NFS implementation

- **Virtualized the file system with *vnodes***
  - Basically poor man's C++ (like `protosw` struct)
- **Vnode structure represents an open (or openable) file**
- **Bunch of generic "vnode operations":**
  - lookup, create, open, close, getattr, setattr, read, write, fsync, remove, link, rename, mkdir, rmdir, symlink, readdir, readlink, ...
  - Called through function pointers, so most system calls don't care what type of file system a file resides on
- **NFS implements vnode operations through *Remote Procedure Calls* (RPC)**
  - Client request to server over network, awaits response
  - Each system call may require a series of RPCs
  - System mostly determined by RPC [RFC 1831] *Protocol*

Stateless operation

- **NFS version 2 protocol specified in [RFC 1094]**
- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **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
- **Can this really work?**

Stateless operation

- **NFS version 2 protocol specified in [RFC 1094]**
- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **Requests are self-contained**
- **Requests are** *mostly* **idempotent**
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- **Can this really work?**
  - Of course, FS not stateless – it stores files
  - E.g., *mkdir* can't be idempotent – second time dir exists
  - But many operations, e.g., *read*, *write* are idempotent

Semantics

- **Attach remote file system on local directory**
  - `mount server:/server/path /client/path`
  - Hard mount – if server unavailable, keep trying forever
  - Soft mount – if server unavailable, time out and return error
- **Component-by-component file name lookup**
- **Authenticate client, assume same users as server**
- **Open files should be usable even if unlinked**
  - Kludge: client just renames the file
- **Permissions usually checked when files opened**
  - So if user owns file but no write perms, allow write anyway
- **Cache consistency**
  - With multiple clients, some departure from local FS semantics

NFS version 3

- **Same general architecture as NFS 2**
- **Specified in RFC 1813 (subset of Open Group spec)**
  - Based on XDR protocol specification language [RFC 1832]
  - XDR defines C structures that can be sent over network; includes tagged unions (to know which union field active)
  - Protocol defined as a set of Remote Procedure Calls (RPCs)
- **New access RPC**
  - Supports clients and servers with different uids/gids
- **Better support for caching**
  - Unstable writes while data still cached at client
  - More information for cache consistency
- **Better support for exclusive file creation**

NFSv3 File handles

```
struct nfs_fh3 {
  /* XDR notation for variable-length array
   * with 0-64 opaque bytes: */
  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
- **Handle generally *doesn't* contain filename**
  - Clients may keep accessing an open file after it's renamed

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** $\langle$directory handle, filename$\rangle \rightarrow$ handle
  - Client walks hierarchy one file at a time
  - No symlinks or file system boundaries crossed
  - Client must expand symlinks

Create

```
struct create3args {
  diropargs3 where;
  createhow3 how;
};

union createhow3 switch (createmode3 mode) {
case UNCHECKED:
case GUARDED:
  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?*

Write discussion

- **When is it okay to lose data after a crash?**
  - *Local file system?*
    If no calls to *fsync*, OK to lose 30 seconds of work after crash
  - *Network file system?*

Write discussion

- **When is it okay to lose data after a crash?**
  - *Local file system?*
    If no calls to *fsync*, OK to lose 30 seconds of work after crash
  - *Network file system?*
    What if server crashes but not client?
    Application not killed, so shouldn't lose previous writes
- **NFSv2 addresses problem by having server write data to disk before replying to a write RPC**
  - Caused performance problems
- **Could NFS2 clients just perform write-behind?**
  - Implementation issues – used blocking kernel threads on write
  - Semantics – how to guarantee consistency after server crash
  - Solution: small # of pending write RPCs, but write through on close; if server crashes, client keeps re-writing until acked

NFSv2 write call

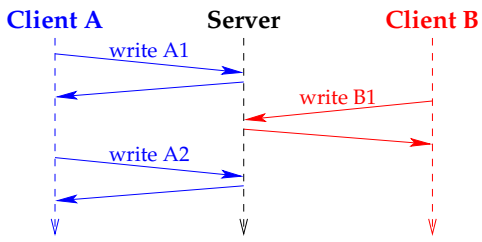
```
struct writeargs {
  fhandle file;
  unsigned beginoffset;
  unsigned offset;
  unsigned totalcount;
  nfsdata data;
};
```

```
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 NFSv2 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)

NFSv3 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
};
```

- **Two goals for NFSv3 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

```
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;
};
```

- **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
  - With NFSv2, would require cache to be flushed
- **With NFSv3, "`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 `stat`ting a file

NFS 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**

NFS version 4 [RFC 3530]

- **Much more complicated than version 3**
  - NFS2: 27 page spec, NFS3: 126 pages, NFS4: 275 pages, NFS4.1: 617 pages
- **Designed to run over higher-latency networks**
  - Support for multi-component lookups to save RTTs
  - Support for batching multiple operations in one RPC
  - Support for leases (in two slides) and stateful (open, close) operation
- **Designed to be more generic and less Unix-specific**
  - E.g., support for extended file attributes, etc.
- **Lots of security stuff**
- **NFS 4.1 [RFC5661] has better support for NAS**
  - Store file data and metadata in different places

Callbacks

- **NFSv2 and v3 poll server for cache consistency**
  - Client requests attributes (via ACCESS) when file opened
  - Attributes validate or invalidate cached copy of file
- **Alternative: Server calls back to clients caching file**
  - Invalidate immediately, rather than when cache needed
  - Requires server to maintain list of all clients caching info
- **Advantages**
  - Tight consistency
- **Disadvantages**
  - Server must maintain a lot of state
  - Updates potentially slow – must wait for n clients to acknowledge
  - When a client goes down, other clients will block

Leases

- **Hybrid mix of polling and callbacks**
  - Server agrees to notify client of changes for a limited period of time – the lease term
  - After the lease expires, client must poll for freshness
- **Avoids paying for a server round trip in many cases**
- **Server doesn't need to keep long-term track of callbacks**
  - E.g., lease time can be shorter than crash-reboot—no need to keep callbacks persistently
- **If client crashes, resume normal operation after lease expiration**

Cache location

- **Can cache on client's local disk, or just mem**
- **Advantages of disk caches**
  - Disk bigger than memory; larger cache → better hit rate
  - Cache persists across a client reboot
- **Potential disadvantages**
  - Uses up disk space on the client
  - Local disk often slower to access than server's memory
  - Have to worry about recovering cache after a crash (wouldn't want to think corrupt cache is latest data)
- **Cooperative caching (xFS, Shark)**
  - With disk or mem cache, clients can fetch data from each other
  - Lessens load on server, can be faster over WAN

Stateful file service (E.g., CIFS)

- **Mechanism:**
  - Client opens a file
  - Server returns client-specific identifier like a file descriptor
  - Identifier used for subsequent accesses until the session ends
  - Server keeps active identifiers in memory; must reclaim
- **Possible advantages**
  - Easier for server to detect sequential access and read ahead
  - Easier to implement callbacks if know all clients w. open file
  - Easier to implement local FS semantics (e.g., unlink file open on different server)
- **Disadvantages**
  - Harder to recover from server crash (lost open file state)

File replication

- **Replicate same file on failure-independent machines**
  - Improves availability and can shorten read time
- **Naming scheme maps file name → good replica**
  - Existence of replicas should be invisible to higher levels
  - Replicas must be distinguished from one another by different lower-level names
- **Updates**
  - Replicas of a file denote the same logical entity
  - Updates must be reflected on all replicas of a file
- **Demand replication – reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica**

AFS interface

- **AFS not designed to replace local disk**
  - E.g., no diskless AFS workstations
- **Instead, provides all remote files under `/afs`**
  - E.g., `/afs/cs.stanford.edu`, `/afs/athena.mit.edu`, ...
- **Each directory under `/afs` corresponds to a *cell***
  - Cells are large administrative entities
  - `CellServDB` file maps all cell names to IP addresses
  - Initially download `CellServDB` from local server
- **Other interesting features**
  - Access control per directory, not per file; can have ACLs
  - `@sys` in symbolic links expands to client's OS
  - Mount points live in the file system, not on the server (`fs mkmount` attaches a volume to a remote directory)

AFS Prototype (AFS 2)

- **Architecture**
  - One server process per client, mirrors AFS files on local disk
- **Protocol:**
  - Files referred to by full pathname
  - Opens go over wire to ensure consistency (verify timestamp)
  - "Stub" directories redirect clients to another server
- **Caching:**
  - File cache keeps whole files
  - Attribute cache for stat calls
- **Results:**
  - Client caching effective
  - Server CPU was bottleneck (context switches, paging, namei)
  - Hard to migrate users' directories to less loaded servers

Current AFS (AFS 3)

- **Cache management**
  - Added directory and symlink caches
  - Added callbacks for invalidation
- **Name resolution**
  - Files named by fid rather than pathname
  - fid is 96 bits: ⟨volume#, vnode#, uniquifier⟩
  - No explicit location information in fid!
- **Process structure**
  - Use LWPs instead of processes (Basically non-preemptive threads)
- **Low-level storage represenation**
  - Change system call interface: `iopen` (today called `fhopen` or `open_by_handle_at`)

AFS caching

- **Directories**
  - Consider request component by component (like NFS)
  - Used cached component if there is callback on directory
  - Otherwise, update directory (if needed) and establish callback
- **Files**
  - Writes are only visible on client workstation
  - On close, changes are flushed back to server
  - All metadata/attribute changes are synchronous
- **Potential complications?**

AFS caching

- **Directories**
  - Consider request component by component (like NFS)
  - Used cached component if there is callback on directory
  - Otherwise, update directory (if needed) and establish callback
- **Files**
  - Writes are only visible on client workstation
  - On close, changes are flushed back to server
  - All metadata/attribute changes are synchronous
- **Potential complications?**
  - Disk full/server error (will only be noticed on close)
  - Access checking (what if one user of client doesn't have permission)
  - Reading first byte of enormous file slow (required fetching whole file – now fixed with *chunking*)

AFS volumes

- **Allow many *volumes* per disk**
  - E.g., each user's home directory might be a volume
- **Quotas established per volume**
- **Read-only volumes can be replicated on multiple servers**
- **Snapshots/backups performed per volume**
  - Using cheap copy-on-write snapshots
- **Volume → server mapping replicated on all servers**

Volume migration

- **Make a *clone* on old server**
  - Cheap, copy-on-write snapshot of volume
- **Copy the clone to the new server**
- **Make a second clone of volume**
  - Copy changes since first clone to new server
- **Freeze volume, copy any final changes**
- **Change volume → server mapping**
  - Old server has forwarding pointer
  - Redirects clients while volume info propagates