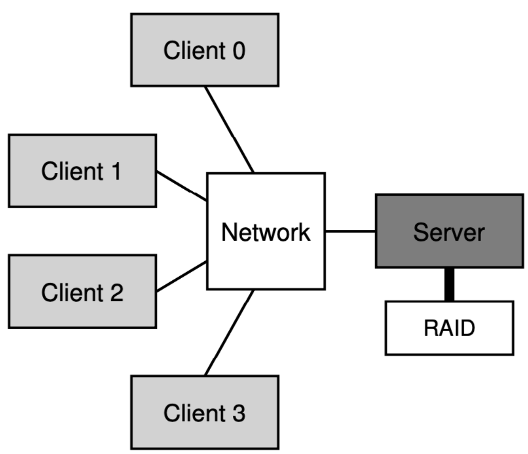
Distributed Systems

- *48 - Communication Basics*
- 49 - NFS
- 50 - AFS
- GFS

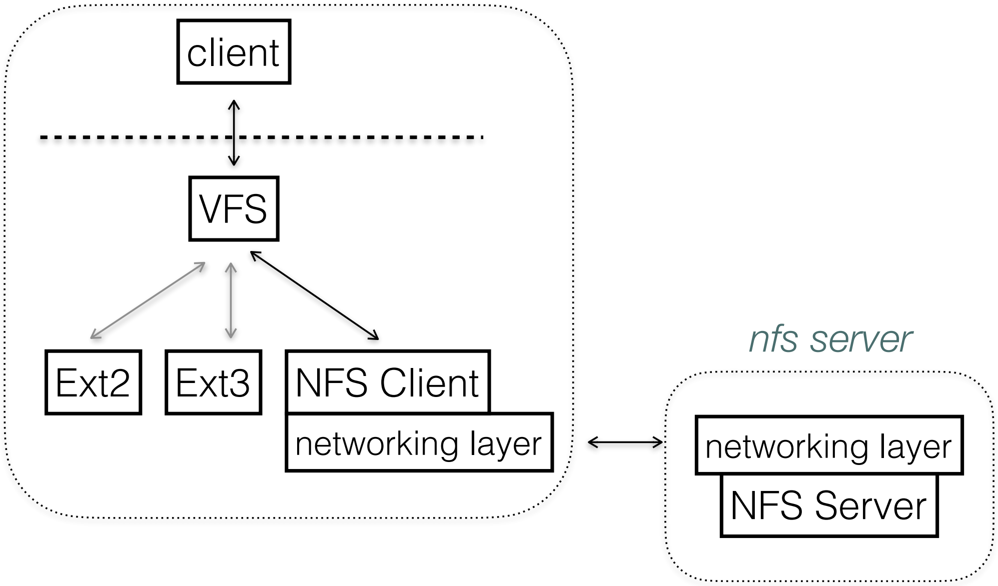
NFS *Sun Microsystems*

- first widely used distributed file system
    - clients diskless
        - easy sharing
        - centralized admin
        - security



NFS

- distributed file system should be *transparent*
  - except possibly in performance
  - client issues same file-system calls as standalone system



NFS *actually NFSv2*

"a distributed system is one where a machine I've never heard of goes down and I can't read my email"

- Leslie Lamport: Turing Award Winner for his work on distributed systems

- NFS goals:
  - simple and fast file recovery
  - *stateless* protocol : *server keeps no client state*
    - server scales well
    - client crashes transparent
    - server crashes transparent
    - client must maintain all state the the server needs for any communication

NFS *actually NFSv2*

- file handle : uniquely describe file or directory
  - volume ID
  - inode number
  - generation number (inumbers get re-used)

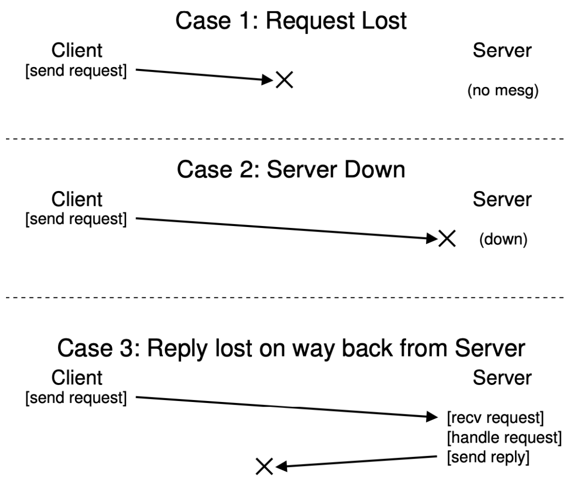
`NFSPROC_GETATTR`	file handle returns: attributes
`NFSPROC_SETATTR`	file handle, attributes returns: –
`NFSPROC_LOOKUP`	directory file handle, name of file/dir to look up returns: file handle
`NFSPROC_READ`	file handle, offset, count data, attributes
`NFSPROC_WRITE`	file handle, offset, count, data attributes
`NFSPROC_CREATE`	directory file handle, name of file, attributes –
`NFSPROC_REMOVE`	directory file handle, name of file to be removed –
`NFSPROC_MKDIR`	directory file handle, name of directory, attributes file handle
`NFSPROC_RMDIR`	directory file handle, name of directory to be removed –
`NFSPROC_READDIR`	directory handle, count of bytes to read, cookie returns: directory entries, cookie (to get more entries)

NFS *reading a file :*

Client	Server
fd = open("/foo", ...); Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	
read(fd, buffer, MAX); Index into open file table with fd get NFS file handle (FH) use current file position as offset Send READ (FH, offset=0, count=MAX)	
	Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client
Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app	
read(fd, buffer, MAX); Same except offset=MAX and set current file position = 2*MAX	
read(fd, buffer, MAX); Same except offset=2*MAX and set current file position = 3*MAX	
close(fd); Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)	

NFS *server failures*

- server crashes / restarts, knowing nothing about clients
  - because most client requests are *idempotent*
    - lookups, reads don't change server state
    - writes contain data and exact offset to write to
- client handles all timeouts in the same way



NFS *performance*

- client-side caching
  - read file data (and metadata) cached by client
  - all good unless the file changes on the server
- client-side write buffers
  - coalescing
  - aggregating disparate messages
  - writes sent back to server asynchronously (but before close())

- However : cache consistency!

NFS *cache consistency*

Problems:

- update visibility
  - C_1 writes `foo.c`, but does not immediately push to server
  - C_2 reads, sees old version
  - C_1 flushes to server
- stale cache
  - C_2 closes and reads again, sees old version (`foo.c` locally cached)

Fixes:

- close-to-open consistency
  - every open guaranteed to see every prior write to the server
    - must validate cache (GETATTR)
    - but maybe not all the time

NFS consistency is weak… *(so are most other FSs)*

NFS *server caching*

- tons of memory
  - wants to use it for disk cache (satisfy reads)
  - wants to use it for write buffer (quickly ack writes)
    - what could go wrong?
- server could ack a write before writing to disk!
  - say file initially has three 4k blocks of data:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

  - client overwrites with:

```
write(aaa…, 0)., write(bbb…, 4k), write(ccc…, 8k):
```

    - server crashes after acking second block, before writing:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops
ccccccccccccccccccccccccccccccccccccccccccc
```

  - client never evens knows that the server crashed

NFS *cache consistency*

Problem: *poor performance for $client_i$ the same file again*

- fix: allow *$client_i$* to cache data and attributes on client
  - *but when $client_i$ re-opens not guaranteed most recent version*
- fix: have clients re-validate on open
  - *but slow*
- fix: time out the cached attributes
  - means data can all be cached, attributes *sometimes* validated w/ server before accesses
  - *but when $client_i$ re-opens not guaranteed most recent version (still)*

non-fix: *NFS consistency is weak… (same true for other FS's)*

NFS *innovations*

- stateless protocol
  - minimizes state server needs to track
  - server can crash and recover w/o clients being aware
- itempotent requests
  - necessary for statelessness
  - client treats network message drops, server failure the same
  - client does not need to know which is which
- client and server buffering
  - essential for performance
  - cache consistency issues
    - server flushes writes before acking
    - client attribute cache times out
- VFS interface
  - makes application API independent of underlying FS

NFS *later versions*

- version v3
  - 64-bit sizes and offsets (large files)
  - synchronous server writes
  - readdirplus (reads dir, also includes the file handles)
  - tcp
- version v4
  - strong security (kerberos, public key protocols)
  - performance improvements
  - stateful protocol (mostly for file consistency)
  - open standard (IETF)

Distributed Systems

- *48 - Communication Basics*
- *49 - NFS*
- 50 - AFS
- Review

Andrew File System *AFS v1*

- primary motivation was *scale*
  - how many clients could a single server accommodate?
  - user-visible behavior well-defined
  - whole-file (not block) caching

```
TestAuth       Test whether a file has changed
               (used to validate cached entries)
GetFileStat    Get the stat info for a file
Fetch          Fetch the contents of file
Store          Store this file on the server
SetFileStat    Set the stat info for a file
ListDir        List the contents of a directory
```

Figure 50.1: **AFSv1 Protocol Highlights**

Andrew File System *AFS v2*

- Problems w/ v1:
  - full path traversal costs (on the server!)
  - client issues too many TestAuth msgs
  - also:
    - load not balanced across servers (fix using volumes)
    - server has a process per client (fix using threads)
- Improving the protocol:
  - client callbacks:
    - promise from the server to notify client if cached file changed
  - file identifier (FID)
    - volume id
    - file id
    - "uniquifier" (usually called epochs elsewhere)

AFS
example

Client (C_1)	Server
fd = open("/home/remzi/notes.txt", ...); Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir establish callback(C_1) on remzi return remzi's content and FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir establish callback(C_1) on notes.txt return notes.txt's content and FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX); perform local `read()` on cached copy	
close(fd); do local `close()` on cached copy if file has changed, flush to server	

fd = open("/home/remzi/notes.txt", ...);
Foreach dir (home, remzi)
if (callback(dir) == VALID)
use local copy for lookup(dir)
else
Fetch (as above)
if (callback(notes.txt) == VALID)
open local cached copy
return file descriptor to it
else
Fetch (as above) then open and return fd	

Andrew File System *cache consistency*

Mentioned two issues w/ NFS:

- update visibility
  - when will server be updated w/ client write?
- cache staleness:
  - when will clients be informed their versions are out of date?

- AFS procedure:
  - client writes, possibly many times
  - *closes*
    - writes complete file back to server, becomes visible
    - server *breaks callback*
      - contact each server w/ a callback and invalidate its copy

all apps on single machine see same copy

Andrew File System *cache consistency*

P$_1$	P$_2$	Client$_1$ Cache	P$_3$	Client$_2$ Cache	Server Disk	*Comments*
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() $\rightarrow$ A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() $\rightarrow$ B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() $\rightarrow$ A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() $\rightarrow$ B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P$_3$
		D	read() $\rightarrow$ D	D	D	the last writer wins
		D	close()	D	D	

Andrew File System *cache consistency*

- AFS provides also *close-to-open* consistency
  - whole-file caching and updating
    - never see concurrent writes diff clients in same version of a file
  - "last writer wins" (really last *closer* wins)
- Crash recovery complicated
  - crashing client might miss callback (*client treats cache as suspect after crash*)
  - crashing server loses callbacks table
    - server might inform all clients after recovery
    - or clients constantly check for server liveness w/ heartbeats
  - *there is a cost to building a more sensible and scalable caching model*

NFS vs AFS

- primarily differ in caching
  - What to cache?
    - NFS caches blocks
    - AFS entire files (on disk)
  - When to push writes to server?
    - Loosely defined for NFS:
      - any time from *right away*, to *when file is closed*
      - (only modified blocks)
    - If any part modified, AFS pushes entire file at `close()`
  - Final contents after concurrent merges by different clients:
    - NFS: writes by the different clients might be intermingled
    - AFS: final version reflects *the last write*; other write is lost

Exam 2 *review*

- exam topics:
  - disk performance: perf from latencies
  - disk scheduling: SSTF, CSCAN, SCAN
  - RAID 0, 1, 5
  - FFS: advantages, write order
  - journaling, meta-data journaling
  - SSDS: simple mapping table, hybrid mapping table,
  - end-to-end argument
  - LFS: structure, write cost calculations
  - NFS and AFS: structure, cache consistency
- what to review
  - quizzes 7-10
  - lectures after spring break

RH 7 *review*

Q1
5 Points

Given disk:

- 6000 RPM
- 200 sectors/track
- sector is 8KB
- avg. seek time 2 msec
- read/write no difference
- no track caching

Write only a number in each of the boxes for this question: no explanation, no units, no nothing.

Q1.1
1 Point

In msecs, what is the average rotational latency?

5

Explanation
6000 RPM $\Rightarrow$ 100/sec $\Rightarrow$ 10 msec per rotation. Average latency would be half of this, i.e. **5** msecs.

Q1.2
1 Point

In msecs, what is the average sector transfer time?

0.05

Explanation
10 msec/rot, 1/200th of a rotation

Q1.3
1 Point

In msecs, what is the average cost of a random 4k read?

7.05

Explanation
$= \text{seek} + \text{latency}_{rotational} + \text{transfer}_{sector}$
$= 2 + 5 + 0.05 = 7.05$ msec
you have to read an entire sector

RH 7 *review*

Q1.5
1 Point

In msecs, what would be the minimum expected cost of reading 10 sequentially ordered sectors?

7.5

Explanation
The minimum expected would be if they were all laid out in the same track, so we only pay seek time and rotational latency once. After that we just pay the transfer time for the rest of the sectors.
$7.05 + 9 * 0.05 = 7.5$ msec

RH 8 *review*

Q2
5 Points

List the writes that should occur when creating a 100-byte file `bar.c` in the directory `/foo` for a generic, non-journaling file system, in a correct order (there may be more than one):

Explanation

write data block bitmap (async)
write `bar.c` data (async)
write inode bitmap (async)
write `bar.c` inode
write `/foo` data
write `/foo` inode

Q3
5 Points

Instead, how many disk writes would a log-structured file system ideally issue?

(enter just an integer)

1

done 4/16