Distributed Filesystems

Continued

Consequences of statelessness

- Read and writes must specify their start offset
  - Server does not keep track of current position in the file
  - User still use conventional UNIX reads and writes
- Open system call translates into several lookup calls to server
- No NFS equivalent to UNIX close system call

Important pieces of protocol

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

From protocol to distributed file system

- Client side translates user requests to protocol messages to implement the request remotely
- Example:

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	

The lookup call (I)

- Returns a **file handle** instead of a file descriptor
  - File handle specifies unique location of file
    - Volume identifier, inode number and generation number
- **lookup(dirfh, name)** *returns* **(fh, attr)**
  - Returns file handle **fh** and attributes of named file in directory **dirfh**
  - Fails if client has no right to access directory **dirfh**

The lookup call (II)

- One single open call such as

  fd = open("/usr/joe/6360/list.txt")

  will be result in several calls to lookup

  lookup(rootfh, "usr") returns (fh0, attr)
  lookup(fh0, "joe") returns (fh1, attr)
  lookup(fh1, "6360") returns (fh2, attr)
  lookup(fh2, "list.txt") returns (fh, attr)

- Why all these steps?
  - Any of components of /usr/joe/6360/list.txt could be a *mount point*
  - Mount points are *client dependent* and mount information is kept above the lookup() level

Server side (I)

- Server implements a write-through policy
  - Required by statelessness
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes

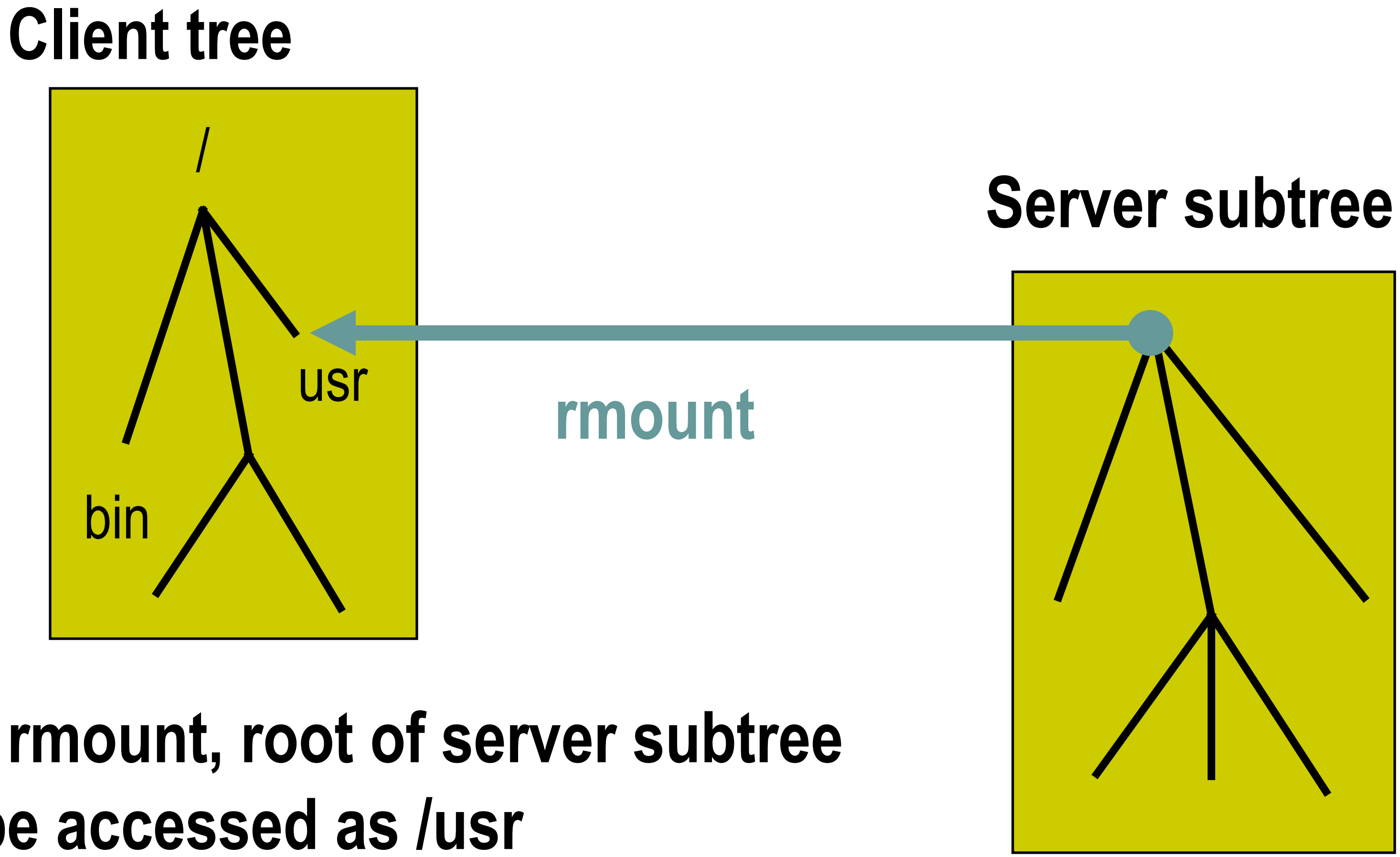
Server side (II)

- File handle consists of
  - Filesystem id identifying disk partition
  - I-node number identifying file within partition
  - Generation number changed every time i-node is reused to store a new file
- Server will store
  - Filesystem id in filesystem superblock
  - I-node generation number in i-node

Client side (I)

- Provides transparent interface to NFS
- Mapping between remote file names and remote file addresses is done a server boot time through ***remote mount***
  - Extension of UNIX mounts
  - Specified in a ***mount table***
  - Makes a remote subtree appear part of a local subtree

Remote mount

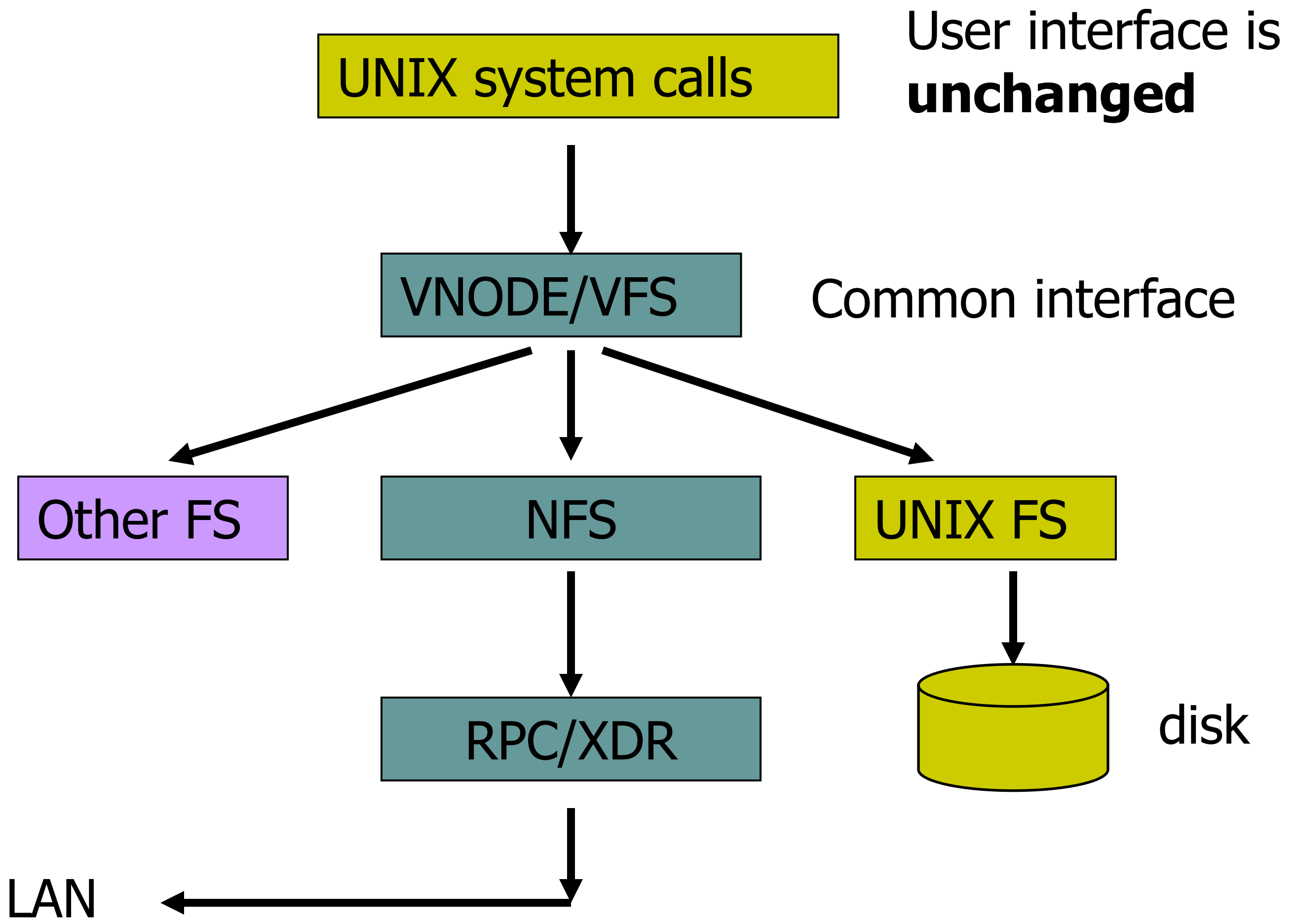


After rmount, root of server subtree can be accessed as /usr

Client side (II)

- Provides transparent access to
  - NFS
  - Other file systems (including UNIX FFS)
- New virtual filesystem interface supports
  - VFS calls, which operate on whole file system
  - VNODE calls, which operate on individual files
- Treats all files in the same fashion

Client side (III)



More examples

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

Continued

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)

Handling server Failures

- Failure types:

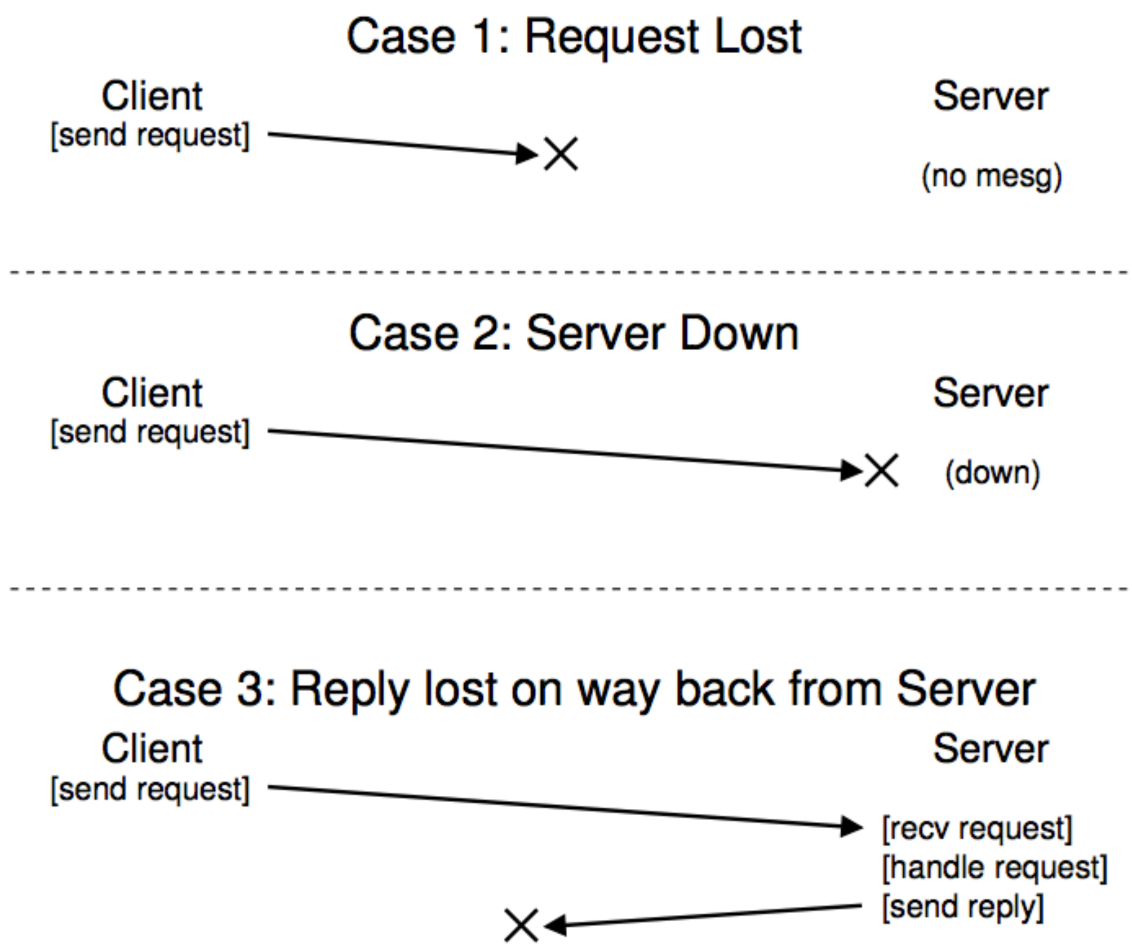


Figure 48.6: **The Three Types of Loss**

Recovery in Stateless NFS

- If the server fails and restarts, there is no need to rebuild in-memory state on the server.
  - Client reestablishes contact (e.g., TCP connection).
  - Client retransmits pending requests.
- Classical NFS uses a connectionless transport (UDP).
  - Server failure is transparent to the client; no connection to break or reestablish.
    - A crashed server is indistinguishable from a slow server.
  - Sun/ONC RPC masks network errors by retransmitting a request after an adaptive timeout.
    - A dropped packet is indistinguishable from a crashed server.

Drawbacks of a Stateless Service

- The stateless nature of classical NFS has compelling design advantages (simplicity), but also some key drawbacks:
  - Recovery-by-retransmission constrains the server interface.
    - ONC RPC/UDP has *execute-at-least-once* semantics (“send and pray”), which compromises performance and correctness.
  - Update operations are disk-limited.
    - Updates *must commit synchronously* at the server.
  - NFS cannot (quite) preserve local *single-copy semantics.*
    - Files may be removed while they are open on the client.
    - Server cannot help in client cache consistency.
- Let’s explore these problems and their solutions...

Problem 1: Retransmissions and Idempotency

- For a connectionless RPC transport, retransmissions can saturate an overloaded server.
  - Clients "kick 'em while they're down", causing steep hockey stick.
- Execute-at-least-once constrains the server interface.
  - Service operations should/must be idempotent.
    - Multiple executions should/must have the same effect.
  - Idempotent operations cannot capture the full semantics we expect from our file system.
    - remove, append-mode writes, exclusive create

Solutions to the Retransmission Problem

- 1. Hope for the best and smooth over non-idempotent requests.
  - E.g., map ENOENT and EEXIST to ESUCCESS.
- 2. Use TCP or some other transport protocol that produces reliable, in-order delivery.
  - higher overhead...and we still need sessions.
- 3. Implement an execute-at-most once RPC transport.
  - TCP-like features (sequence numbers)...and sessions.
- 4. Keep a *retransmission cache* on the server [Juszczak90].
  - Remember the most recent request IDs and their results, and just resend the result....does this violate statelessness?
  - DAFS persistent session cache.

Problem 2: Synchronous Writes

- Stateless NFS servers must commit each operation to stable storage before responding to the client.
  - Interferes with FS optimizations, e.g., clustering, LFS, and disk write ordering (seek scheduling).
    - Damages bandwidth and scalability.
  - Imposes disk access latency for each request.
    - Not so bad for a logged write; much worse for a complex operation like an FFS file write.
- The synchronous update problem occurs for any storage service with reliable update (*commit*).

Speeding Up Synchronous NFS Writes

- Interesting solutions to the synchronous write problem, used in high-performance NFS servers:
  - Delay the response until convenient for the server.
    - E.g., NFS *write-gathering* optimizations for clustered writes (similar to *group commit* in databases).
    - Relies on write-behind from NFS I/O daemons (*iods*).
  - Throw hardware at it: non-volatile memory (NVRAM)
    - Battery-backed RAM or UPS (uninterruptible power supply).
    - Use as an operation log (Network Appliance WAFL)...
    - ...or as a non-volatile disk write buffer (Legato).
  - Replicate server and buffer in memory (e.g., MIT Harp).

NFS V3 Asynchronous Writes

- NFS V3 sidesteps the synchronous write problem by adding a new *asynchronous write* operation.
  - Server may reply to client as soon as it accepts the write, before executing/committing it.
    - If the server fails, it may discard *any subset* of the accepted but uncommitted writes.
  - Client holds asynchronously written data in its cache, and reissues the writes if the server fails and restarts.
    - When is it safe for the client to discard its buffered writes?
    - How can the client tell if the server has failed?

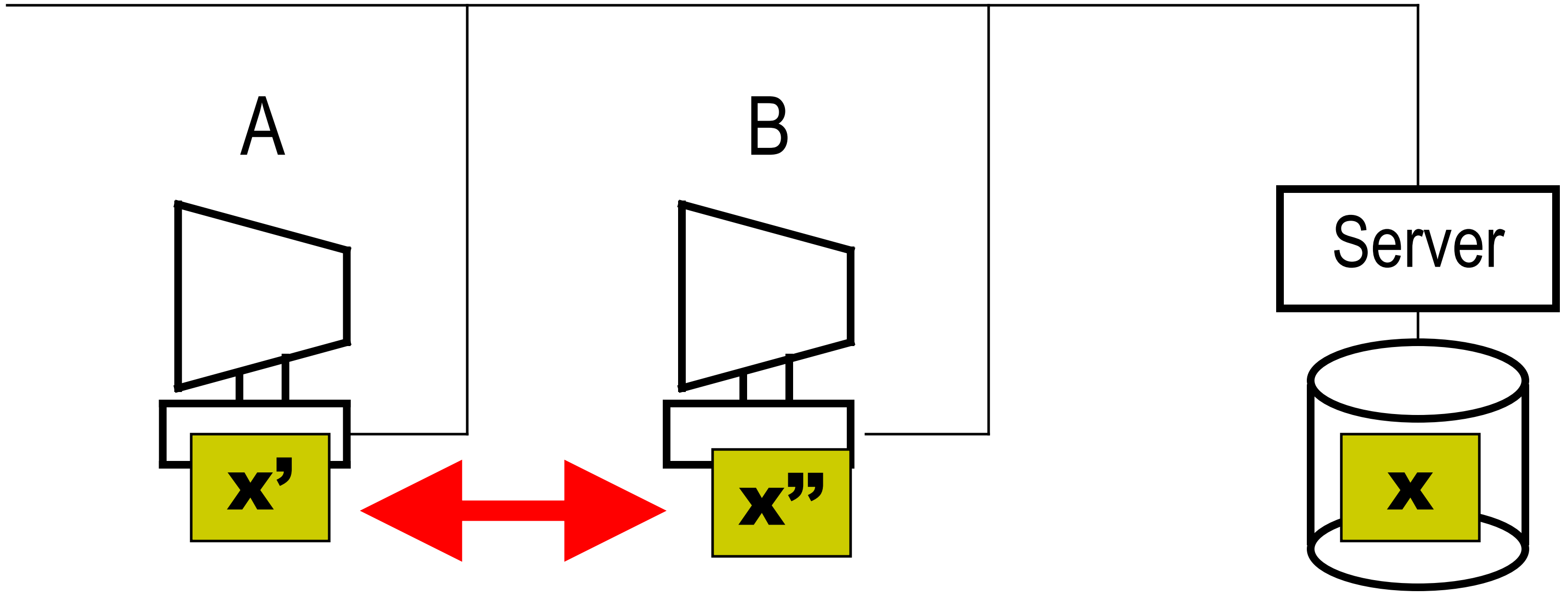
NFS V3 Commit

- NFS V3 adds a new *commit* operation to go with async-write.
  - Client may issue a *commit* for a file byte range at any time.
  - Server must execute all covered uncommitted writes before replying to the commit.
  - When the client receives the reply, it may safely discard any buffered writes covered by the commit.
  - Server returns a *verifier* with every reply to an *async write* or *commit* request.
    - The verifier is just an integer that is guaranteed to change if the server restarts, and to never change back.
  - What if the client crashes?

File consistency/coherence issues

- Cannot build an efficient network file system without ***client caching***
  - *Cannot send each and every read or write to the server*
- ***Client caching introduces** coherence **issues***
- Conventional timeshared UNIX semantics guarantee that
  - All writes are executed in strict sequential fashion
  - Their effect is immediately visible to all other processes accessing the file
- Interleaving of writes coming from different processes is left to the kernel discretion
-

Example



Inconsistent updates X' and X'' to file X

Example

- Consider a one-block file X that is concurrently modified by two workstations
- If file is cached at *both* workstations
  - A will not see changes made by B
  - B will not see changes made by A
- We will have
  - Inconsistent updates
  - Non respect of UNIX semantics

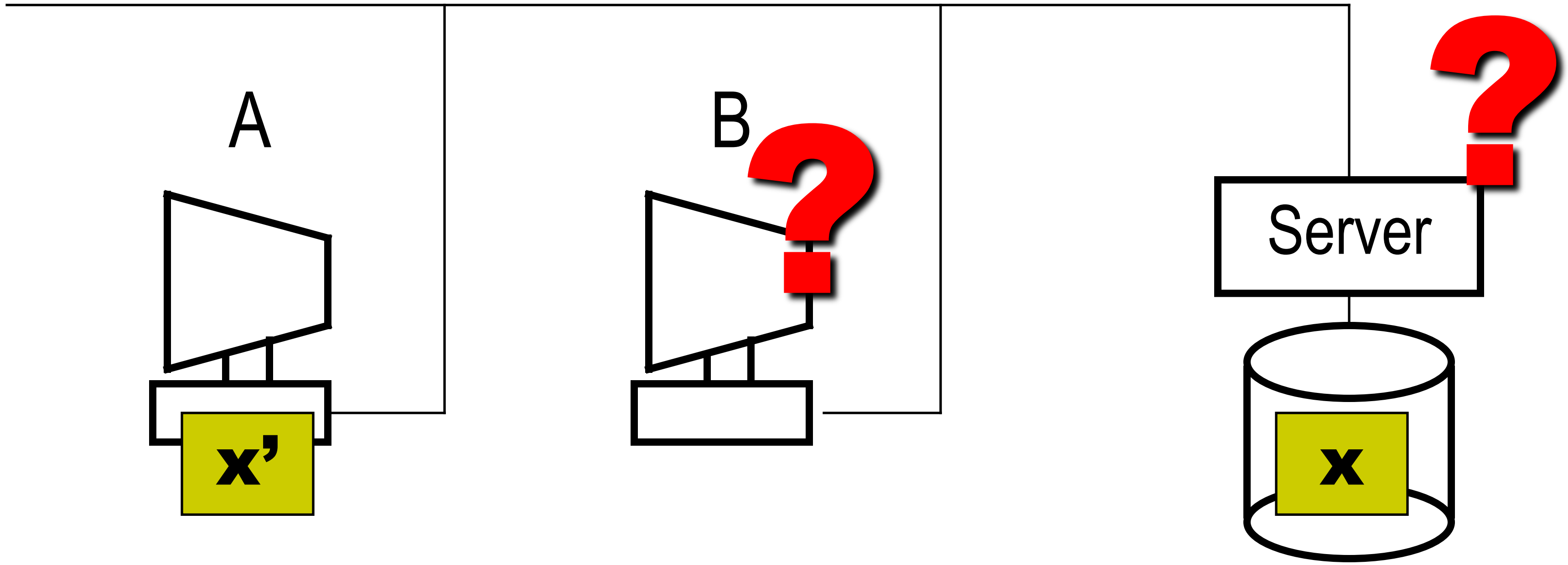
UNIX file access semantics (II)

- UNIX file access semantics result from the use of a single I/O buffer containing all cached blocks and i-nodes
- Server caching is not a problem
- Disabling client caching is not an option:
  - Would be too slow
  - Would overload the file server

NFS solution (I)

- Stateless server does not know how many users are accessing a given file
  - ***Clients do not know either***
- Clients ***must***
  - Frequently send their modified blocks to the server
  - Frequently ask the server to revalidate the blocks they have in their cache

NFS solution (II)



Better to propagate my updates and refresh my cache

Implementation

- VNODE interface only made the kernel 2% slower
- Few of the UNIX FS were modified
- MOUNT was first included into the NFS protocol
  - Later broken into a separate user-level RPC process

Problem 3: File Cache Consistency

- Problem: Concurrent write sharing of files.
  - Contrast with *read sharing* or *sequential write sharing*.
- Solutions:
  - *Timestamp invalidation* (NFS).
    - Timestamp each cache entry, and periodically query the server: "has this file changed since time ***t***?"; invalidate cache if stale.
  - *Callback invalidation* (AFS, Sprite, Spritely NFS).
    - Request notification (callback) from the server if the file changes; invalidate cache and/or disable caching on callback.
  - *Leases* (NQ-NFS) [Gray&Cheriton89,Macklem93,NFS V4]
  - Later: distributed shared memory

File Cache Example: NQ-NFS Leases

- In NQ-NFS, a client obtains a *lease* on the file that permits the client's desired read/write activity.
  - "A lease is a ticket permitting an activity; the lease is valid until some expiration time."
  - A *read-caching lease* allows the client to cache clean data.
    - **Guarantee**: no other client is modifying the file.
  - A *write-caching lease* allows the client to buffer modified data for the file.
    - **Guarantee**: no other client has the file cached.
    - Allows *delayed writes*: client may delay issuing writes to improve write performance (i.e., client has a writeback cache).

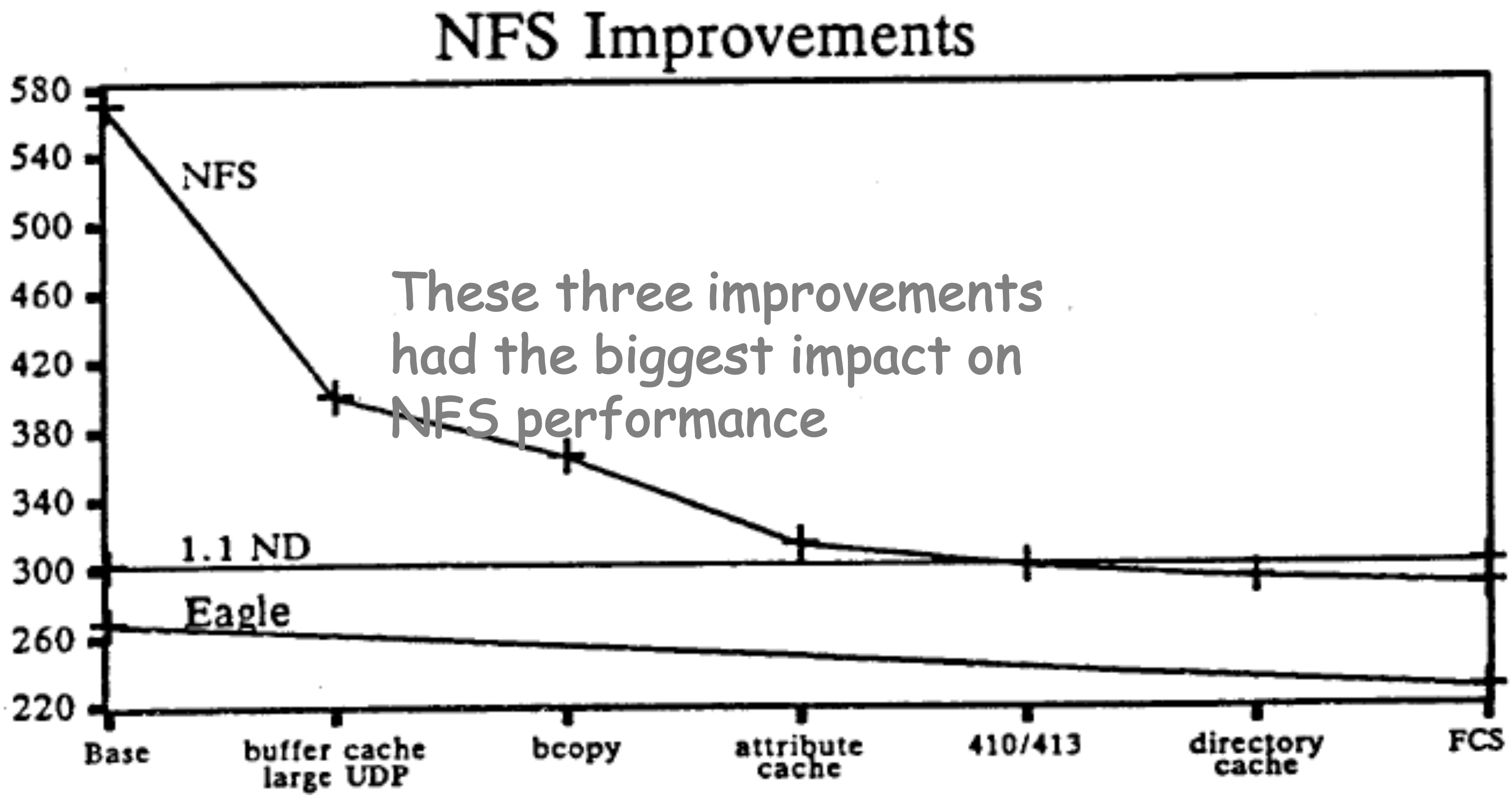
Tuning (I)

- First version of NFS was much slower than Sun Network Disk (ND)
- First improvement
  - Added client buffer cache
  - Increased the size of UDP packets from 2048 to 9000 bytes
- Next improvement reduced the amount of buffer to buffer copying in NFS and RPC (bcopy)

Tuning (II)

- Third improvement introduced a client-side **attribute cache**
  - Cache is updated every time new attributes arrive from the server
  - Cached attributes are discarded after
    - 3 seconds for ***file attributes***
    - 30 seconds for ***directory attributes***
- These three improvements cut benchmark run time by 50%

Tuning (III)



These three improvements had the biggest impact on NFS performance

Conclusions

- NFS succeeded because it was
  - Robust
  - Reasonably efficient
  - Tuned to the needs of diskless workstations

In addition, NFS was able to evolve and incorporate concepts such as close-to-open consistency

Discussion

- Throughput
- Latency
- Scalability
- Crash Recovery
- Fault Tolerance

AFS: Andrew File System

- Main Motivation: Scalability!!!

- Basic idea: whole-file caching
  - Fetch the whole file for the first time
  - Update on close

AFS version 1

- When open a file for the first time, cache it
- Next time, TestAuth to determine if the file has changed

- Performance is poor. Why?
  - Path-traversal costs are too high
  - Too many TestAuth messages

AFS version 2

- Solution
  - File identifier
    - Similar to file handle in NFS
  - A callback mechanism to reduce client/server interactions
    - An analogy to polling vs. interrupts

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	

Client$_1$			Client$_2$		Server	*Comments*
P_1	P_2	Cache	P_3	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → 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() → 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() → D	D	D	the last writer wins
		D	close()	D	D	

AFS Crash Recovery

- If a client crashes, it treats all cache contents as suspect. Send TestAuth to the server.
- If the server crashes, it asks all clients to reconstruct the callback states

Discussion Again

- Throughput
- Latency
- Scalability
- Crash Recovery
- Fault Tolerance