Network File System (NFS)

Brad Karp
UCL Computer Science

CS GZ03 / M030
3rd October 2014

NFS Is Relevant

- Original paper from 1985
- Very successful, still widely used today
- Early result; much subsequent research in networked filesystems “fixing shortcomings of NFS”
- NFS is a great substrate for cool distributed systems NCS projects!

Why Build NFS?

- Why not just store your files on local disk?
- Sharing data: many users reading/writing same files (e.g., code repository), but running on separate machines
- Manageability: ease of backing up one server
- Disks may be expensive (true when NFS built; no longer true)
- Displays may be expensive (true when NFS built; no longer true)

Goals for NFS

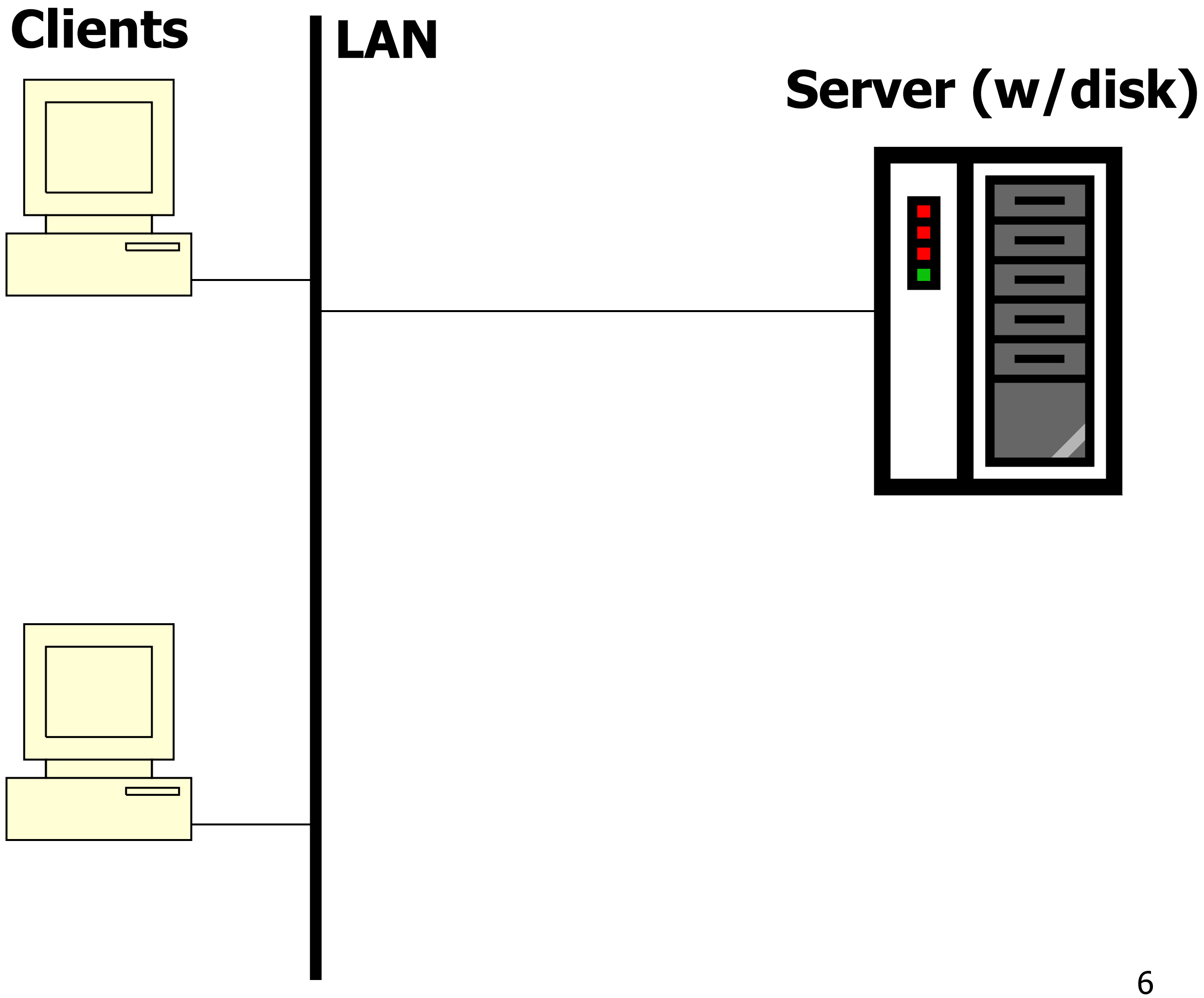
- Work with existing, unmodified apps:
  - Same semantics as local UNIX filesystem
- Easily deployed
  - Easy to add to existing UNIX systems
- Compatible with non-UNIX OSes
  - Wire protocol cannot be too UNIX-specific
- Efficient "enough"
  - Needn't offer same performance as local UNIX filesystem

Goals for NFS

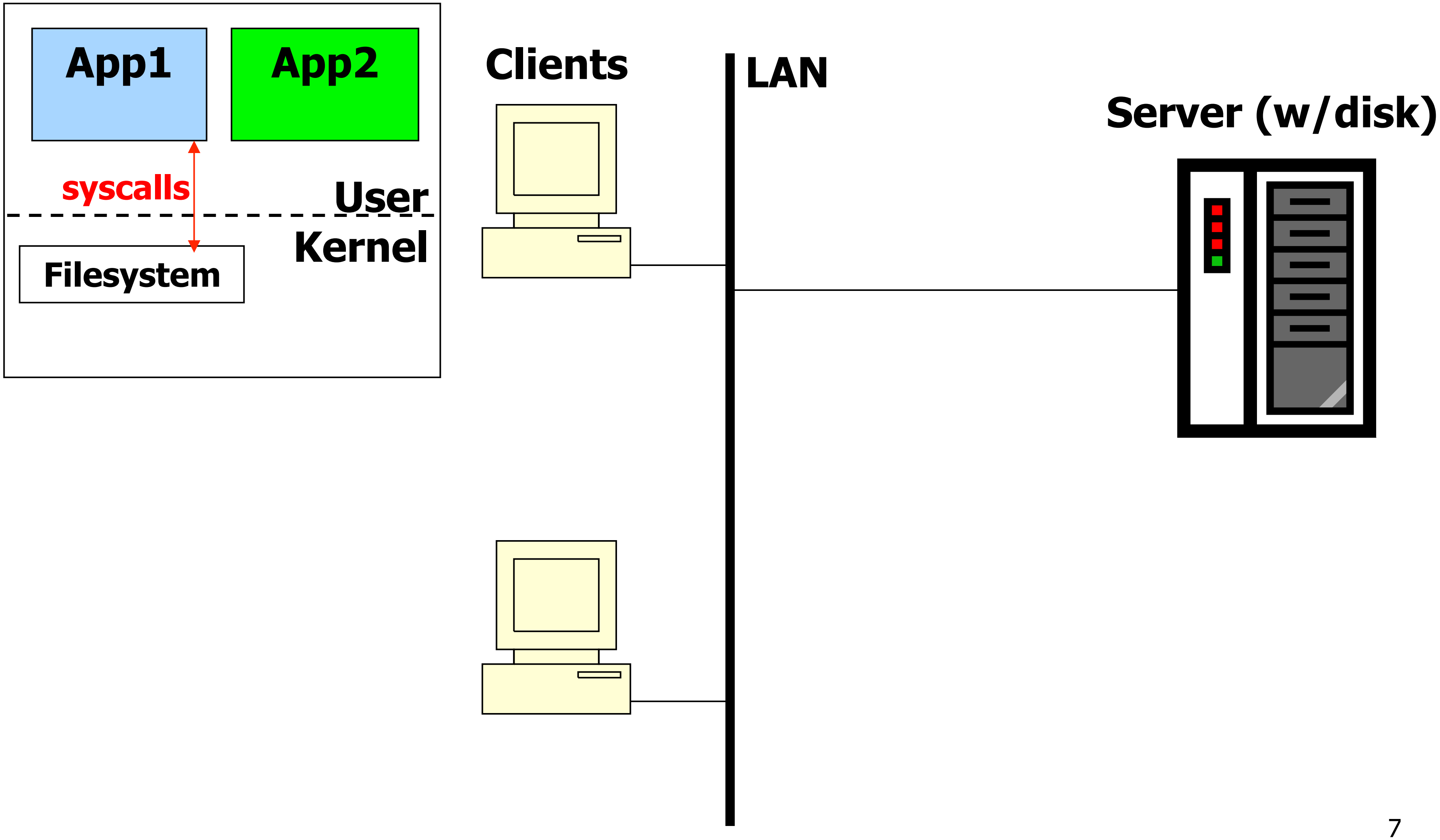
- Work with existing, unmodified apps:
  - Same semantics as local UNIX filesystem
- Easily deployed
  - Easy to add to existing UNIX systems
- Compatible with non-UNIX OSes
  - Wire protocol cannot be too UNIX-specific

Ambitious, conflicting goals!
Does NFS achieve them all fully?
Hint: Recall "New Jersey" approach

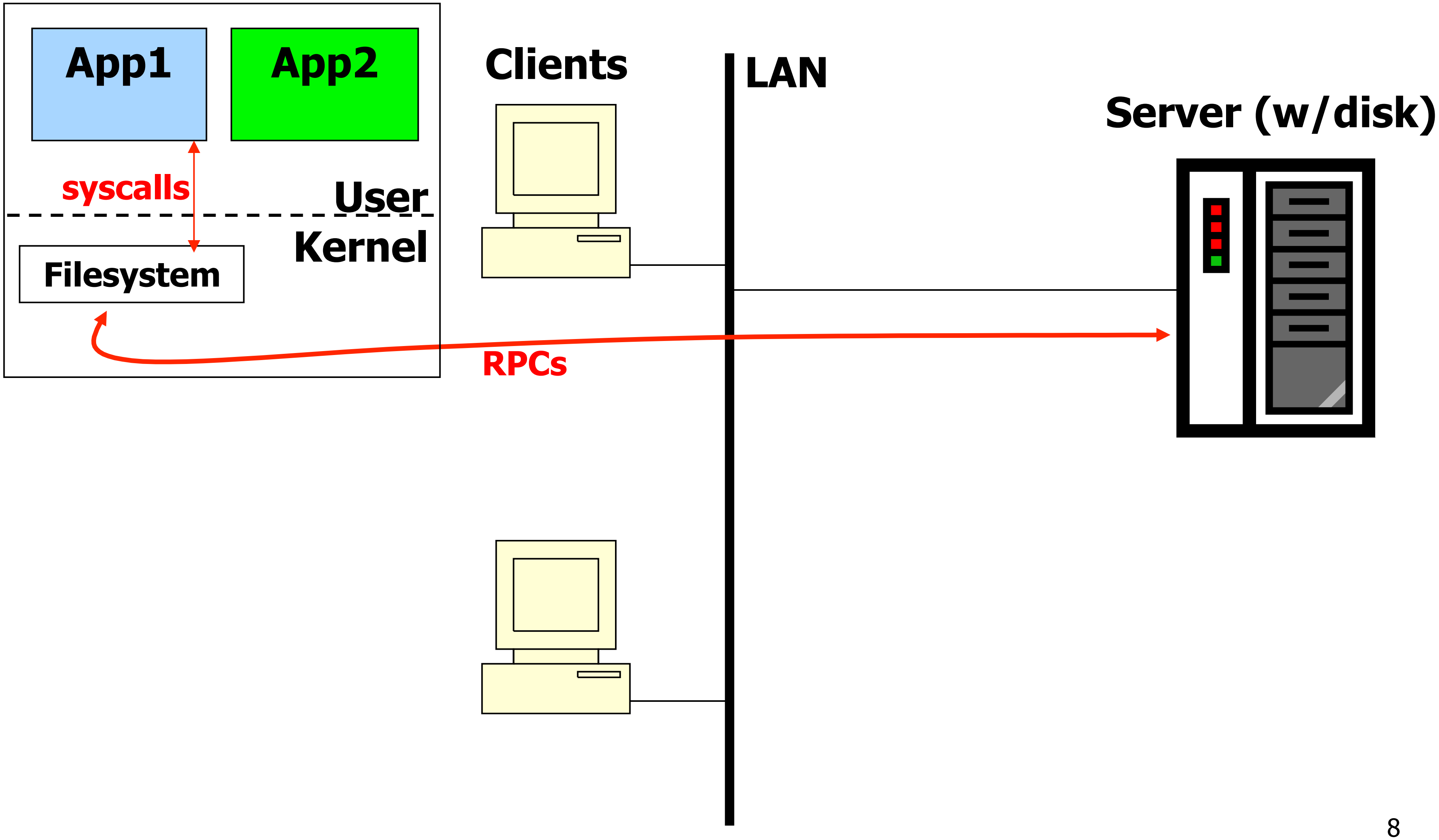
NFS Architecture



NFS Architecture



NFS Architecture



App1

App2

syscalls

User

Kernel

Filesystem

RPCs

Clients

LAN

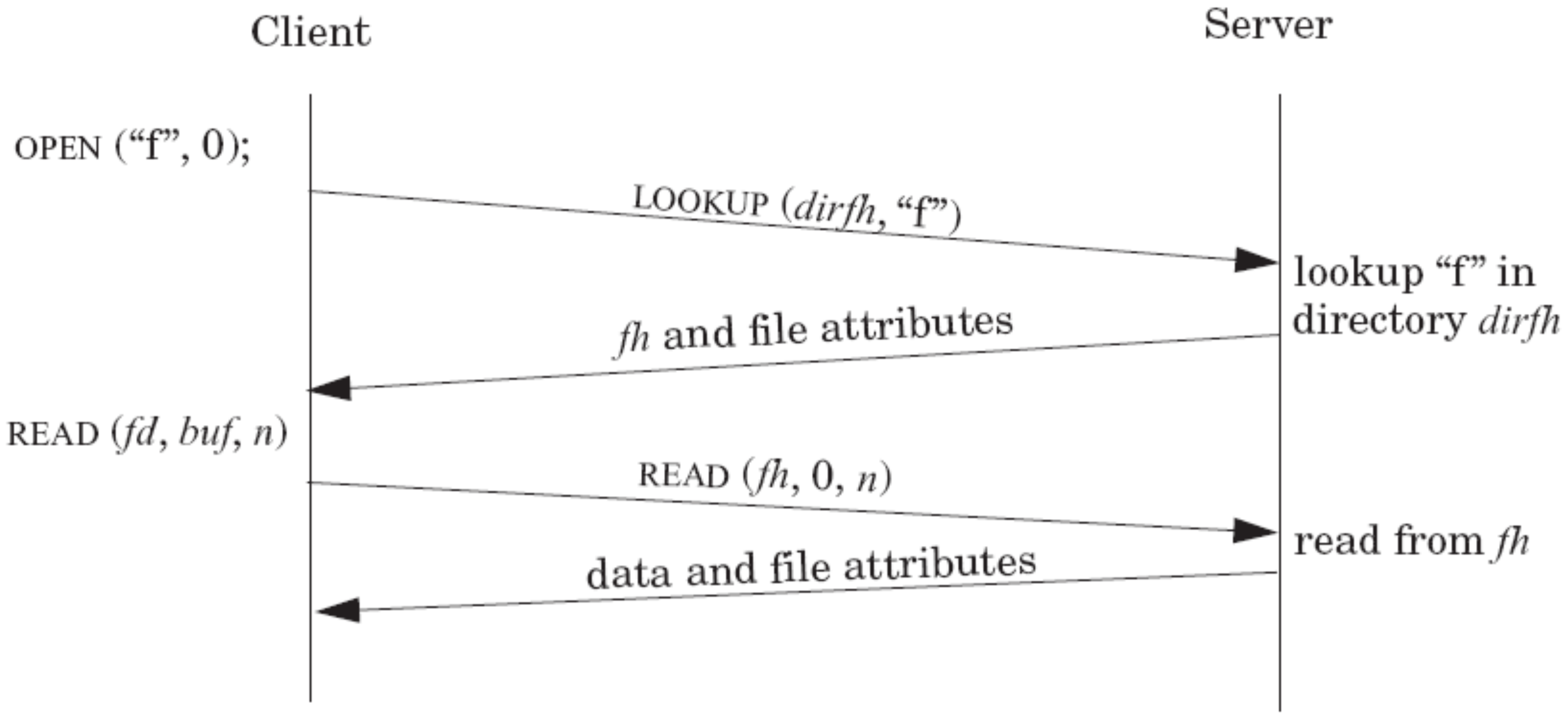
Server (w/disk)

Simple Example: Reading a File

- What RPCs would we expect for:

```
fd = open("f", 0);
read(fd, buf, 8192);
close(fd);
```

Simple Example: NFS RPCs for Reading a File



- Where are RPCs for close()?

File Handle: Function and Contents

- 32-byte name, opaque to client
- Identifies object on remote server
- Must be included in all NFS RPCs
- File handle contains:
  - filesystem ID
  - i-number (essentially, physical block ID on disk)
  - generation number

Generation Number: Motivation

- Client 1 opens file
- Client 2 opens same file
- Client 1 deletes the file, creates new one
- UNIX local filesystem semantics:
  - Client 2 (App 2) sees old file
- In NFS, suppose server re-uses i-node
  - Same i-number for new file as old
  - RPCs from client 2 refer to new file's i-number
  - Client 2 sees new file!

Generation Number: Solution

- Each time server frees i-node, increments its generation number
  - Client 2's RPCs now use old file handle
  - Server can distinguish requests for old vs. new file
- Semantics still not same as local UNIX fs!
  - Apps 1 and 2 sharing local fs: client 2 will see old file
  - Clients 1 and 2 on different workstations sharing NFS fs: client 2 gets error "stale file handle"

Generation Number: Solution

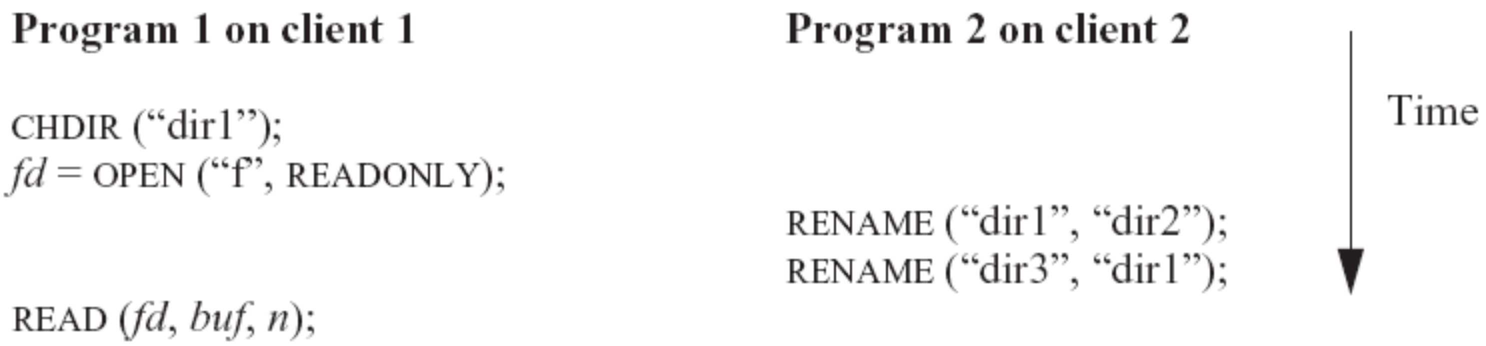
- Each time server frees i-node, increments its generation number

Trade precise UNIX fs semantics for simplicity

New Jersey approach…

- Semantics still not same as local UNIX fs!
  - Apps 1 and 2 sharing local fs: client 2 will see old file
  - Clients 1 and 2 on different workstations sharing NFS fs: client 2 gets error "stale file handle"

Why i-numbers, not Filenames?


Program 1 on client 1

CHDIR ("dir1");
fd = OPEN ("f", READONLY);

READ (*fd*, *buf*, *n*);

Program 2 on client 2

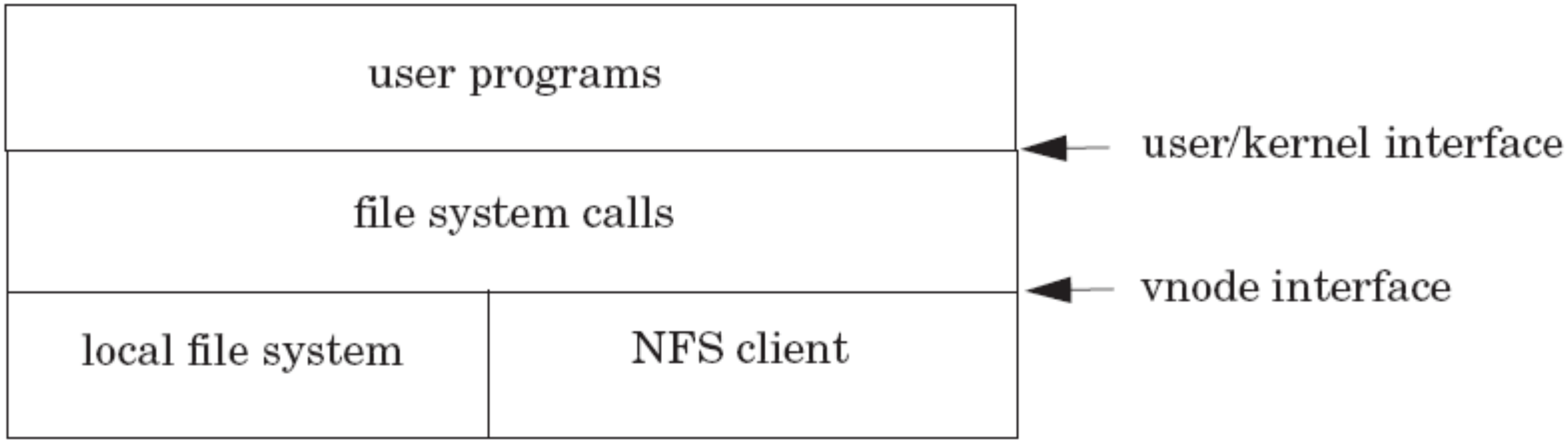
RENAME ("dir1", "dir2");
RENAME ("dir3", "dir1");

- Local UNIX fs: client 1 reads dir2/f
- NFS with pathnames: client 1 reads dir1/f
- Concurrent access by clients can change object referred to by filename
  - Why not a problem in local UNIX fs?
- i-number refers to actual object, not filename

Where Does Client Learn File Handles?

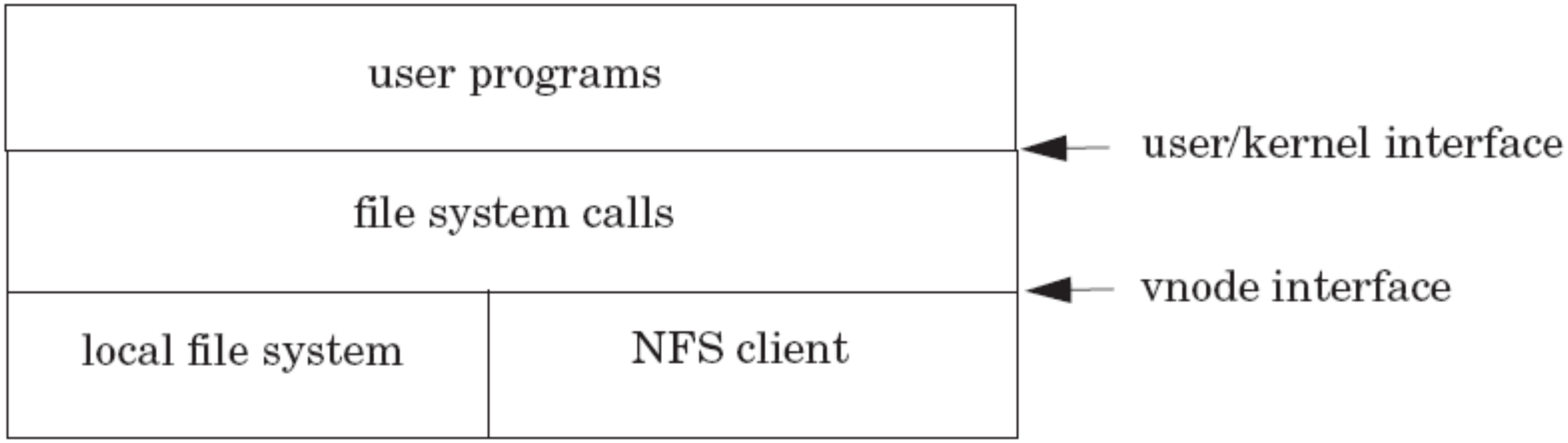
- Before READ, client obtains file handle using LOOKUP or CREATE
- Client stores returned file handle in vnode
- Client’s file descriptor refers to vnode
- **Where does client get very first file handle?**

NFS Implementation Layering



- Why not just send syscalls over wire?
- UNIX semantics defined in terms of files, not just filenames: file’s identity is i-number on disk
- Even after rename, all these refer to same object as before:
  - File descriptor
  - Home directory
  - Cache contents

NFS Implementation Layering



vnode's purpose: remember file handles!

filenames: file's identity is i-number on disk

- Even after rename, all these refer to same object as before:
  - File descriptor
  - Home directory
  - Cache contents

Example: Creating a File over NFS

- Suppose client does:

```
fd = creat(“d/f”, 0666);
write(fd, “foo”, 3);
close(fd);
```

- RPCs sent by client:
  - newfh = LOOKUP (fh, “d”)
  - filefh = CREATE (newfh, “f”, 0666)
  - WRITE (filefh, 0, 3, “foo”)

Server Crashes and Robustness

- Suppose server crashes and reboots
- Will client requests still work?
  - Will client's file handles still make sense?
  - Yes! File handle is disk address of i-node
- What if server crashes just after client sends an RPC?
  - Before server replies: client doesn't get reply, retries
- What if server crashes just after replying to WRITE RPC?

WRITE RPCs and Crash Robustness

- What must server do to ensure correct behavior when crash after WRITE from client?
- Client’s data safe on disk
- i-node with new block number and new length safe on disk
- Indirect block safe on disk
- Three writes, three seeks: 45 ms
- 22 WRITEs/s, so 180 KB/s

WRITEs and Throughput

- Design for higher write throughput:
  - Client writes entire file sequentially at Ethernet speed (few MB/s)
  - Update inode, &c. afterwards
- Why doesn’t NFS use this approach?
  - What happens if server crashes and reboots?
  - Does client believe write completed?
- Improved in NFSv3: WRITEs async, COMMIT on close()

Client Caches in NFS

- Server caches disk blocks
- Client caches file content blocks, some clean, some dirty
- Client caches file attributes
- Client caches name-to-file-handle mappings
- Client caches directory contents
- General concern: what if client A caches data, but client B changes it?

Multi-Client Consistency

- Real-world examples of data cached on one host, changed on another:
  - Save in emacs on one host, “make” on other host
  - “make” on one host, run program on other host
- (No problem if users all run on one workstation, or don’t share files)

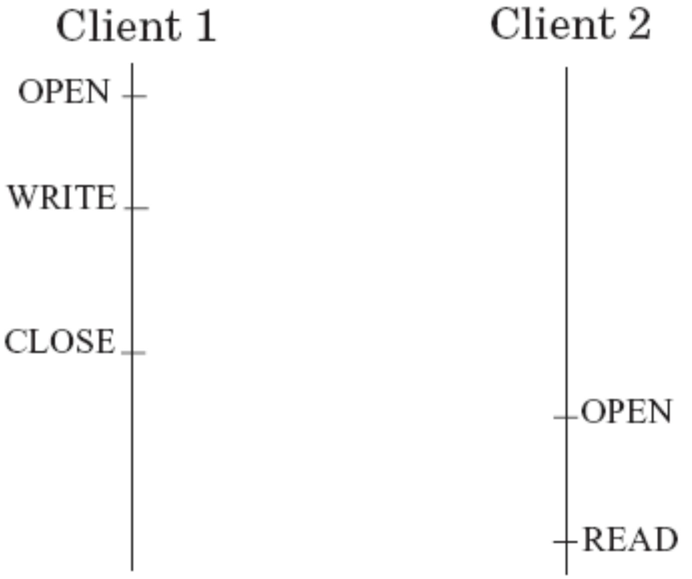
Consistency Protocol: First Try

- On every read(), client asks server whether file has changed
  - if not, use cached data for file
  - if so, issue READ RPCs to get fresh data from server
- Is this protocol sufficient to make each read() see latest write()?
- What's effect on performance?
- Do we need such strong consistency?

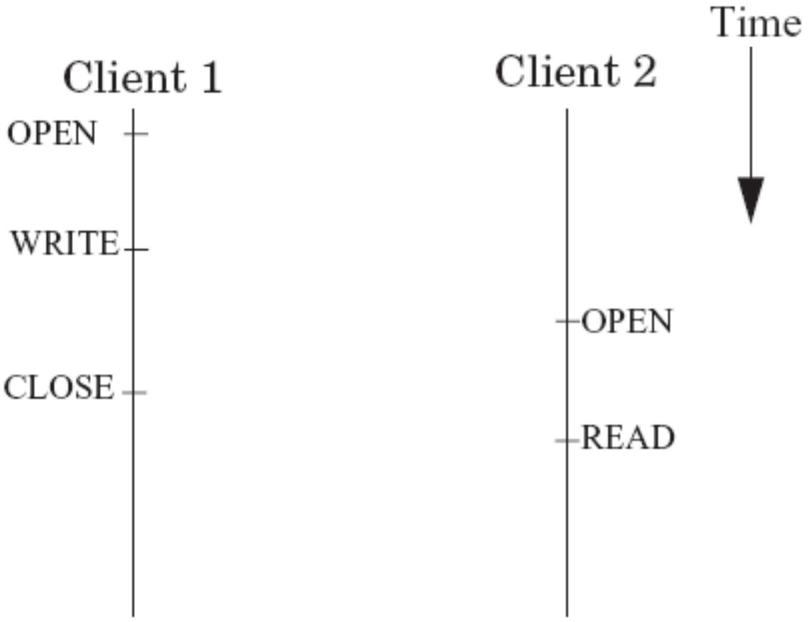
Compromise: Close-to-Open Consistency

- Implemented by most NFS clients
- Contract:
  - if client A write()s a file, then close()s it,
  - then client B open()s the file, and read()s it,
  - client B's reads will reflect client A's writes
- Benefit: clients need only contact server during open() and close()—not on every read() and write()

Compromise:
Close-to-Open Consistency

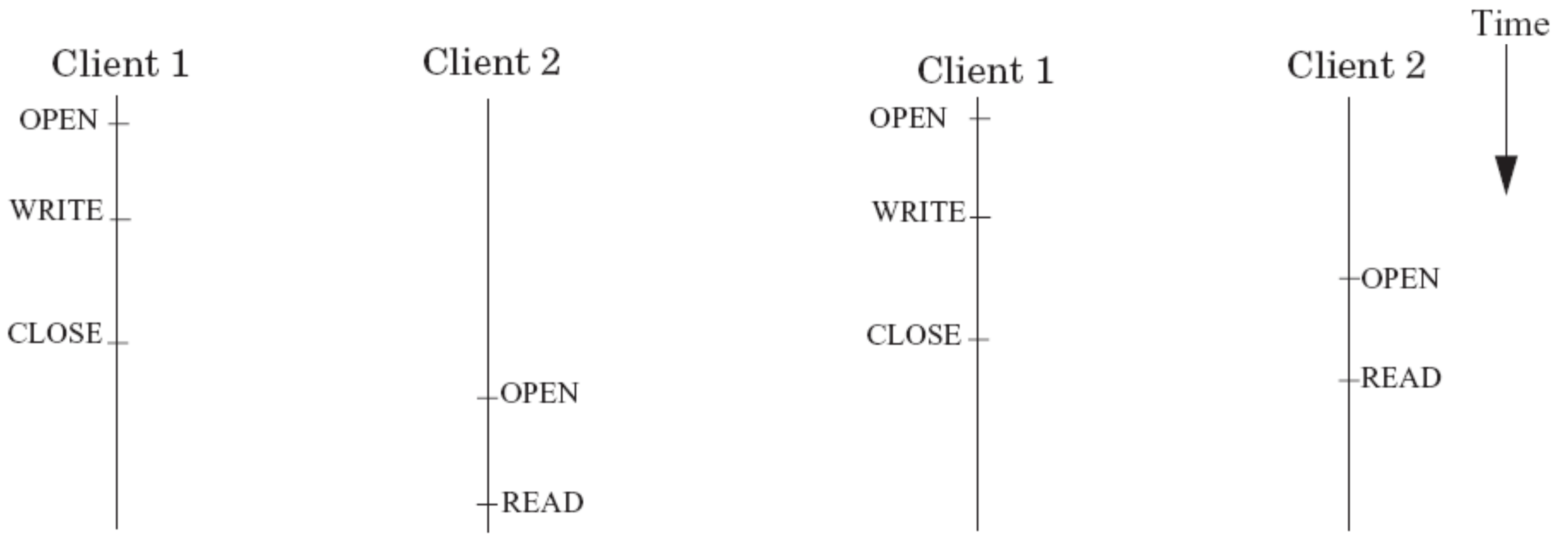


case 1: READ observes last WRITE



case 2: READ may observe last WRITE or not

Compromise:
Close-to-Open Consistency



case 1: READ observes last WRITE

case 2: READ may observe last WRITE or not

Fixes "emacs save, then make" example…

…so long as user waits until emacs says it's done saving file!

Close-to-Open Implementation

- FreeBSD UNIX client (not part of protocol spec):
  - Client keeps file mtime and size for each cached file block
  - close() starts WRITEs for all file’s dirty blocks
  - close() waits for all of server’s replies to those WRITEs
  - open() always sends GETATTR to check file’s mtime and size, caches file attributes
  - read() uses cached blocks only if mtime/length have not changed
  - client checks cached directory contents with GETATTR and ctime

Name Caching in Practice

- Name-to-file-handle cache not always checked for consistency on each LOOKUP
  - If file deleted, may get “stale file handle” error from server
  - If file renamed and new file created with same name, may even get wrong file’s contents

NFS: Secure?

- What prevents unauthorized users from issuing RPCs to an NFS server?
  - e.g., remove files, overwrite data, &c.
- What prevents unauthorized users from forging NFS replies to an NFS client?
  - e.g., return data other than on real server

NFS: Secure?

- What prevents unauthorized users from issuing RPCs to an NFS server?
  - e.g., remove files, overwrite data, &c.
- What prevents unauthorized users from forging NFS replies to an NFS client?
  - e.g., return data other than on real server

IP-address-based authentication of mount requests weak at best; no auth of server to client

Security not a first-order goal in original NFS

Limitations of NFS

- Security: what if untrusted users can be root on client machines?
- Scalability: how many clients can share one server?
  - Writes always go through to server
  - Some writes are to “private,” unshared files that are deleted soon after creation
- Can you run NFS on a large, complex network?
  - Effects of latency? Packet loss? Bottlenecks?

Limitations of NFS

- Security: what if untrusted users can be root on client machines?

Despite its limitations, NFS a huge success:

Simple enough to build for many OSes

Correct enough and performs well enough to be practically useful in deployment

that are deleted soon after creation

- Can you run NFS on a large, complex network?
  - Effects of latency? Packet loss? Bottlenecks?