416 Distributed Systems

Distributed File Systems 1: NFS

Jan 20, 2022

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
    - NFS: network file system
    - AFS: andrew file system
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

Why DFSs are Useful

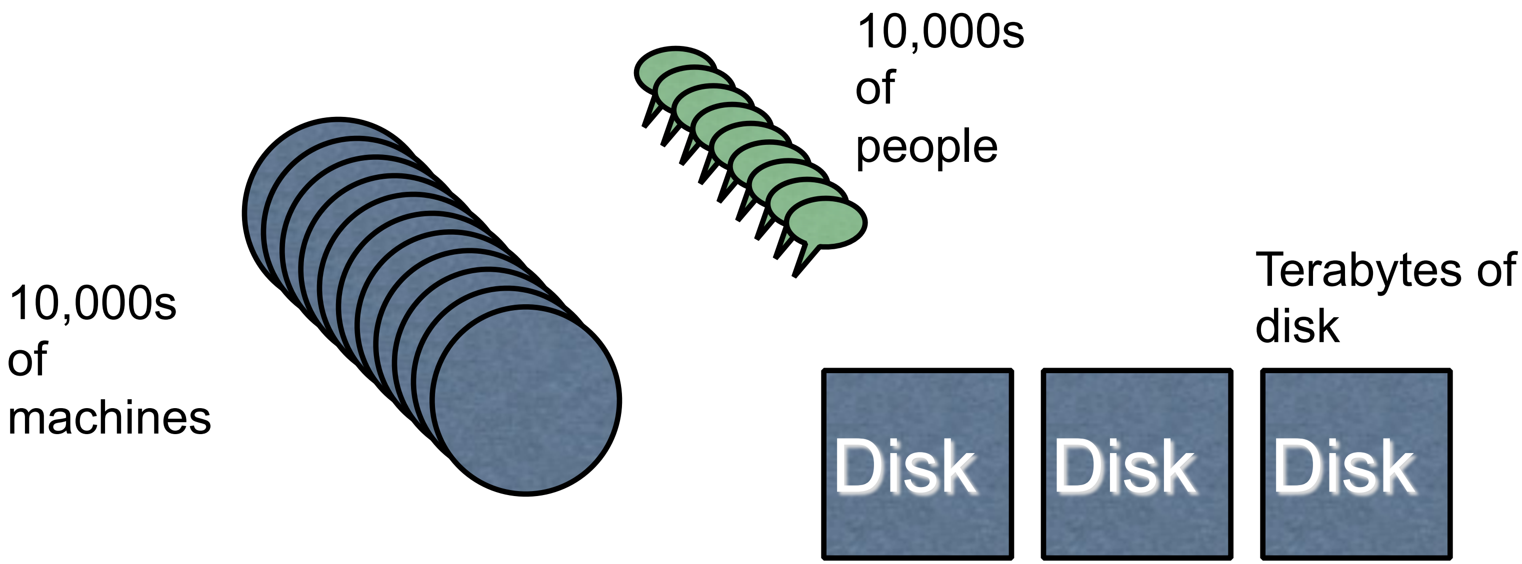
- Data sharing among multiple users
- User mobility
- Location transparency
- Backups and centralized management (security!)

What Distributed File Systems Provide

- Access to data stored at servers using file system interfaces
- What are the file system interfaces?
  - Open a file, check status of a file, close a file
  - Read data from a file
  - Write data to a file
  - Lock a file or part of a file
  - List files in a directory, create/delete a directory
  - Delete a file, rename a file, add a symlink to a file
  - Etc
- (why retain the file system interfaces?)

The andrew file system example

- First example, AFS: developed and used on CMU campus



Goal: Have a consistent namespace for files across computers. Allow any authorized user to access their files from any computer

AFS Challenges

- Remember our initial list of challenges...
- Heterogeneity (lots of different computers & users)
- Scale (10s of thousands of peeps!)
- Security (my files! hands off!)
- Failures
- Concurrency
- oh no... We’ve got ‘em all.

How can we build this??

Just as important: AFS non-challenges

- Geographic distance and high latency
  - AFS targets the campus network, *not* the wide-area

AFS Prioritized goals? / Assumptions

- Often very useful to have an explicit list of prioritized goals. Distributed filesystems almost always involve trade-offs
- Scale, scale, scale
- User-centric workloads... how do users use files (vs. big programs?)
  - Most files are personally owned
  - Not too much concurrent access; user usually only at one or a few machines at a time
  - Sequential access is common; reads much more common that writes
  - There is locality of reference (if you've edited a file recently, you're likely to edit again)

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples (NFS this lecture)
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

Components in a DFS Implementation

- Client side:
  - What has to happen to enable applications to access a remote file the same way a local file is accessed?
  - Accessing remote files in the same way as accessing local files → kernel support

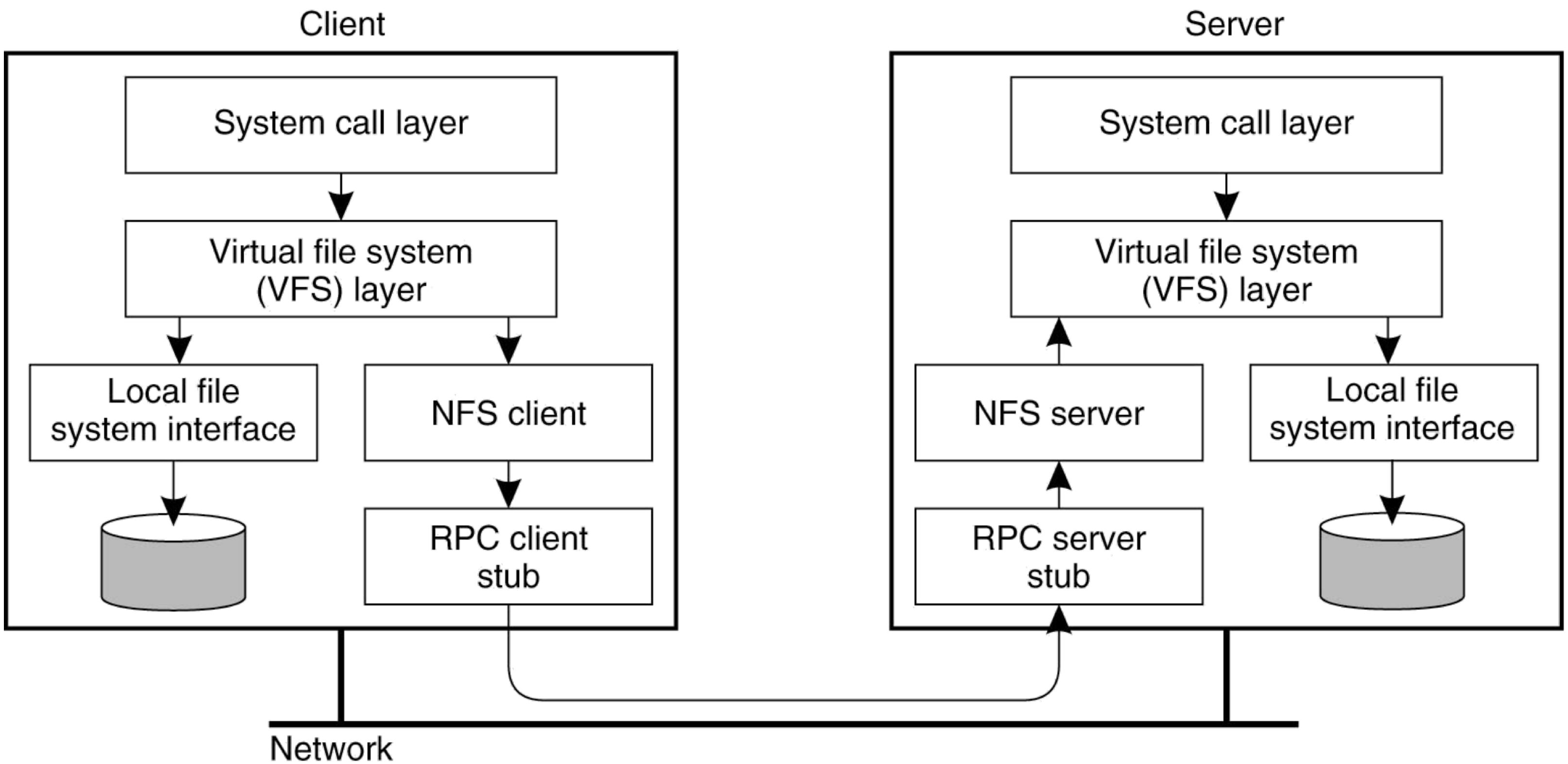
- Communication layer:
  - Just TCP/IP or a protocol at a higher level of abstraction?

- Server side:
  - How are requests from clients serviced?

VFS interception

- VFS provides “pluggable” file systems
- Standard flow of remote access
  - User process calls read()
  - Kernel dispatches to VOP_READ() in some VFS
  - **dfs_read()**
    - check local cache
    - *send RPC to remote Distributed FS server*
    - put process to sleep
  - **server interaction handled by kernel process**
    - retransmit if necessary
    - *convert RPC response to file system buffer*
    - store in local cache
    - wake up user process
  - **dfs_read()**
    - copy bytes to user memory

VFS Interception



A Simple Approach

- Use RPC to forward every FS operation to the server
  - Server serializes all accesses, performs them; sends back result.
- Great: Same behavior as if both programs were running on the same local filesystem! (ignoring latency/failures)
- Bad: Performance can stink. Latency of access to remote server often much higher than to local memory.
- For AFS: bad bad bad: server would get hammered!

Lesson 1: Needing to hit the server for every detail impairs performance and scalability.

Question 1: How can we avoid going to the server for everything? *What* can we avoid this for? What do we lose in the process?

NFS V2 Context and design

- Small number of clients
- Single administrative domain

- “Dumb”, “Stateless” servers w/ smart clients
- Portable across different OSes
- Low implementation cost

- Why a stateless server?

Some NFS V2 RPC Calls

- NFS RPCs using XDR over, e.g., TCP/IP

RPC	Input args	Results
LOOKUP	dirfh, name	status, fhandle, fattr
READ	fhandle, offset, count	status, fattr, data
CREATE	dirfh, name, fattr	status, fhandle, fattr
WRITE	fhandle, offset, count, data	status, fattr

- Key: stateless server!
  - Compare write NFS RPC with local OS syscall write
- fhandle: 32-byte opaque data (64-byte in v3)

Some NFS V2 RPC Calls

- NFS RPCs using XDR over, e.g., TCP/IP

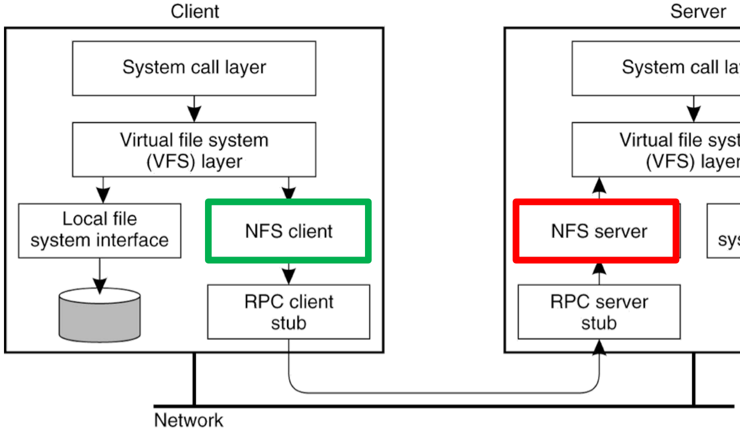
RPC	**Input args**	**Results**
LOOKUP	dirfh, name	status, fhandle, fattr
READ	fhandle, offset, count	status, fattr, data
CREATE	dirfh, name, fattr	status, fhandle, fattr
WRITE	**fhandle, offset, count, data**	**status, fattr**

- Key: stateless server!
  - Compare write NFS RPC with local OS syscall write
- fhandle: 32-byte opaque data (64-byte in v3)

Server Side Example: mountd and nfsd

- mountd: provides the initial file handle for the exported directory
  - Client issues nfs_mount request to mountd
  - mountd checks if the pathname is a directory and if the directory should be exported to the client
- nfsd: answers
- the RPC calls, gets reply from local file system, and sends reply via RPC
  - Usually listening at port 2049

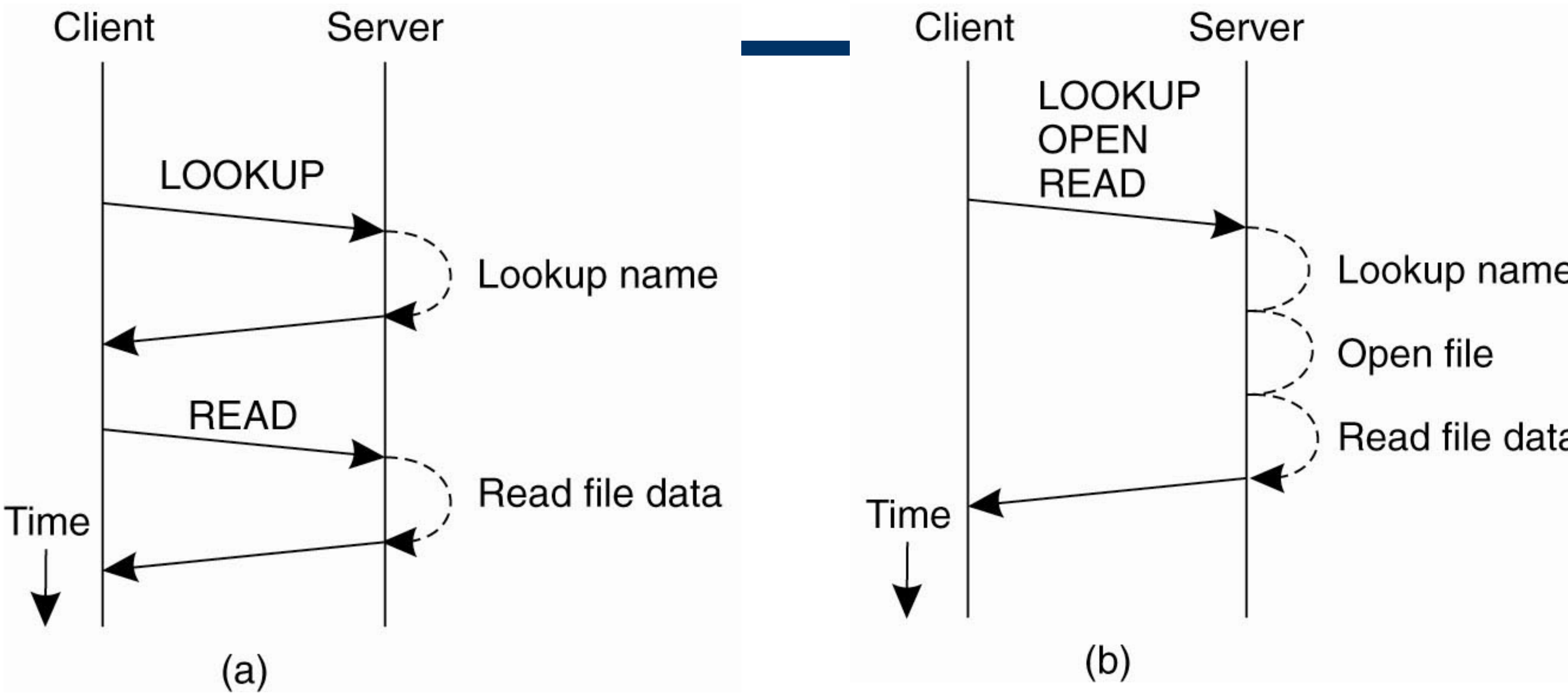
Both mountd and nfsd use underlying RPC implementation



Operator Batching

- Should each client/server interaction accomplish one file system operation or multiple operations?
  - Advantage of batched operations?
- Examples of Batched Operators
  - NFS v3:
    - READDIRPLUS
  - NFS v4:
    - COMPOUND RPC calls

Remote Procedure Calls in NFS



- (a) Reading data from a file in NFS version 3
- (b) Reading data using a compound procedure in version 4.

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

Topic 1: Client-Side Caching

- Many systems (not just distributed!) rely on two solutions to every problem:
    1. Cache it!
    2. *“All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem.”* -- David Wheeler

- Two dist. FS concerns caching helps with:
    - High network load, high server load
    - Surviving failures

Client-Side Caching

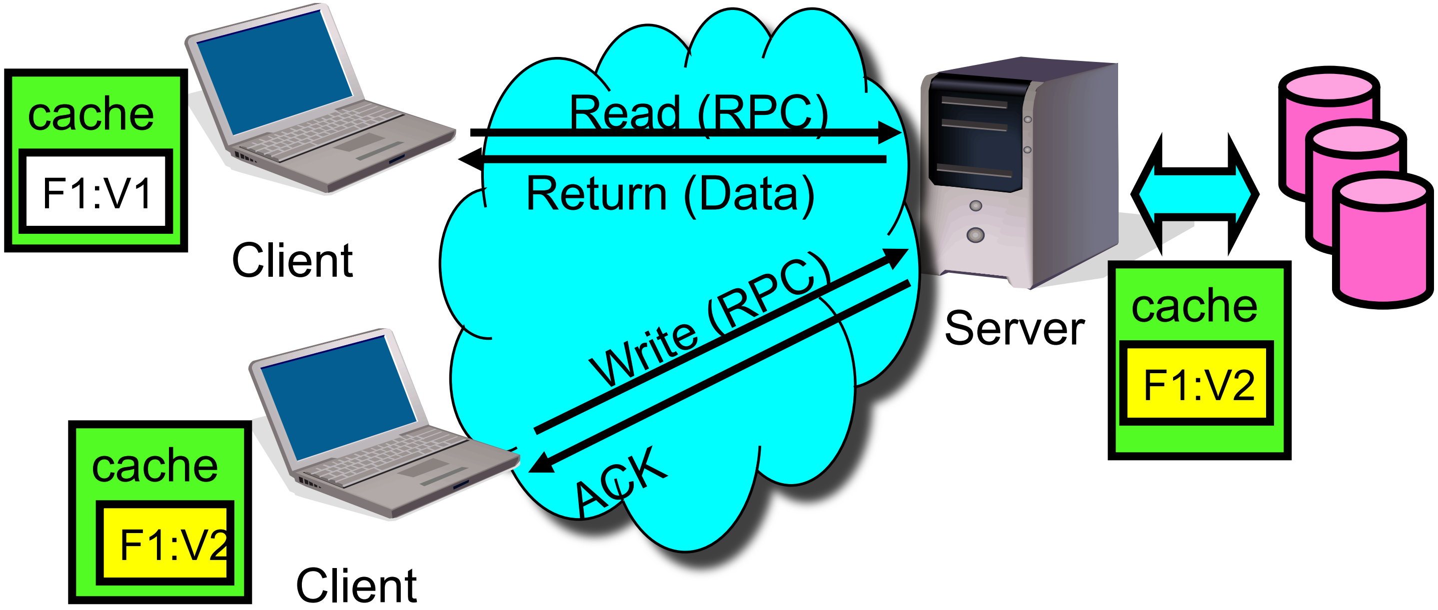
- So, uh, what do we cache?
  - Read-only file data and directory data → easy
  - Data written by the client machine → when is data written to the server? What happens if the client machine goes down?
  - Data that is written by other machines → how to know that the data has changed? How to ensure data consistency?
  - Is there any pre-fetching? (grab before it's needed)
- And if we cache... doesn't that risk making things inconsistent?

Failures

- Server crashes
  - So... what if client does
    - seek() ; /* SERVER CRASH */; read()
    - If server maintains file position, this will fail (Why?). Ditto for open(), read()
  - Or, data in memory, but disk fails
- Lost messages: what if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created foo anew?
- Client crashes
  - Might lose data in client cache

Use of caching to reduce network load (NFS example)

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V1



write(f1)→OK
read(f1)→V2

Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
  - Memory (e.g., DRAM) cache
- File attributes in the client cache expire after 60 seconds (file data doesn't expire)
- File data is checked against the modified-time in file attributes (which could be a cached copy)
  - Changes made on one machine can take up to 60 seconds to be reflected on another machine
- Dirty data are buffered on the client machine until file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost

Implication of NFS v2 Client Caching

- Advantage: No network traffic if open/read/write/close can be done locally.
- But…. Data consistency guarantee is very poor
  - Simply unacceptable for some distributed applications
  - Imagine an application that modifies/reads a lot of shared state across multiple instances (e.g., distributed Game)
- Generally clients do not cache data on local disks

NFS’ s Failure Handling – Stateless Server

- Files are state, but...
- Server exports files without creating extra state
  - No list of “who has this file open” (permission check on each operation on open file!)
  - No “pending transactions” across crash
- Crash recovery is “fast”
  - Reboot, let clients figure out what happened
- State stashed elsewhere
  - Separate MOUNT protocol
  - Separate NLM locking protocol
- Stateless protocol: requests specify exact state. read() → read([file], [position]). no seek on server.

NFS’ s Failure Handling

- Operations are idempotent
  - How can we ensure this?

NFS’ s Failure Handling

- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It’s not delete(“foo”), it’ s delete(1337f00f), where that ID won’t be reused (e.g., by same/other clients)

NFS' s Failure Handling

- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.
- Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Close failures?
    - retry until things get through to the server
    - return failure to client
  - Most client apps can't handle failure of close() call.
  - Usual option: hang for a long time trying to contact server

NFS Take-aways

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it’s gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)