416 Distributed Systems

Distributed File Systems 1

Jan 19, 2018

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

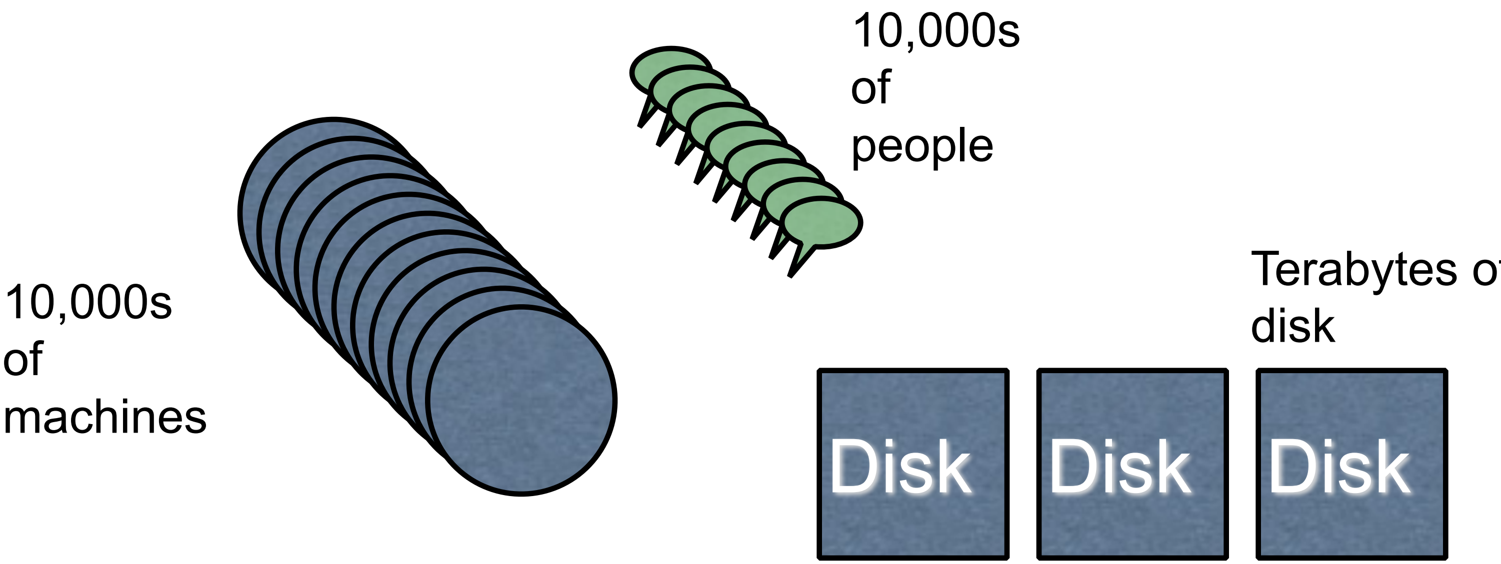
- Data sharing among multiple users
- User mobility
- Location transparency
- Backups and centralized management

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

- 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

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: non-challenges

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

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
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

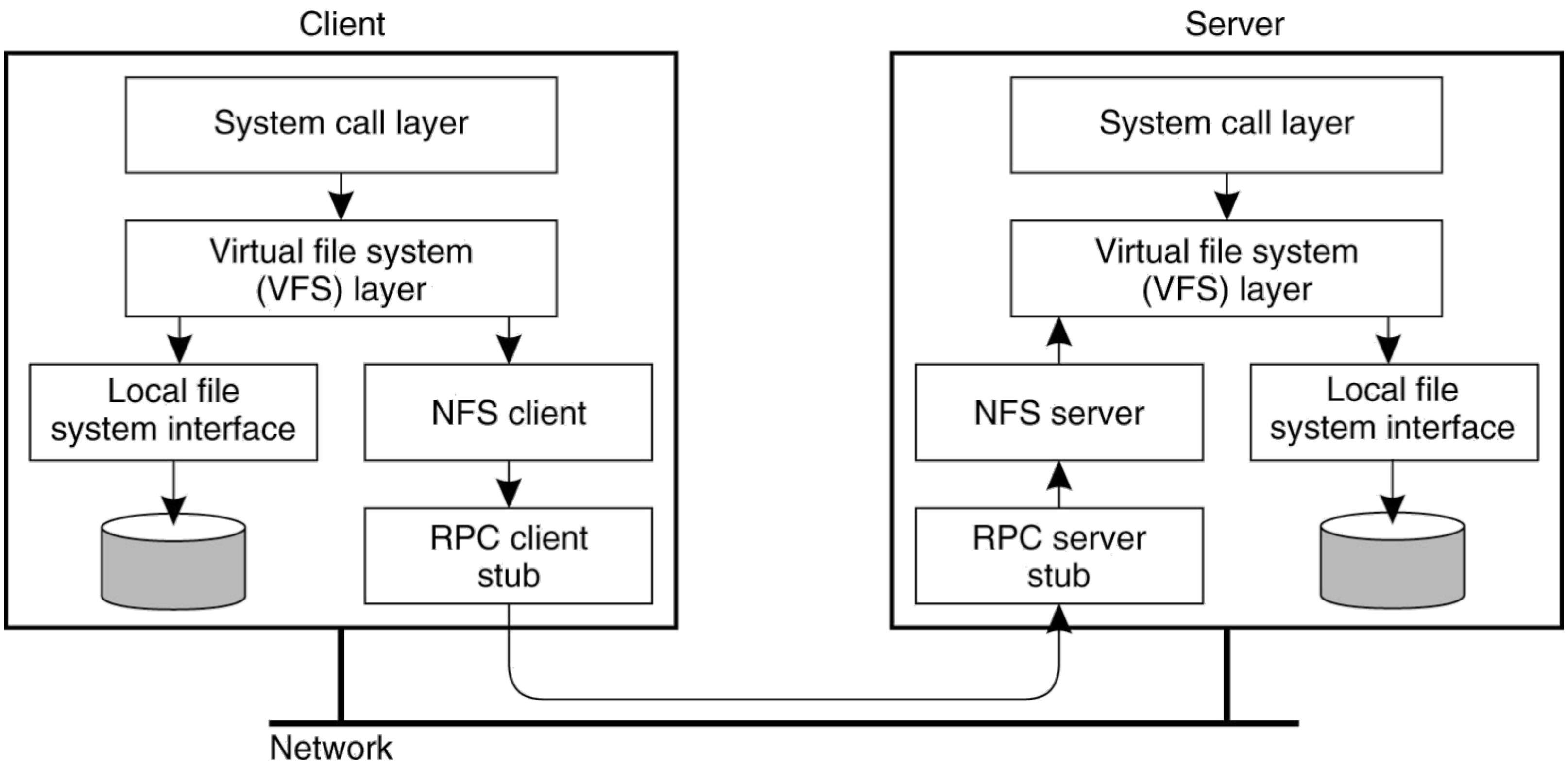
Components in a DFS Implementation

- 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 filesystem operation to the server
  - Server serializes all accesses, performs them, and 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 context: 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

Some NFS V2 RPC Calls

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

Proc.	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!
- 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

NFS V2 Operations

- V2:
  - NULL, GETATTR, SETATTR
  - LOOKUP, READLINK, READ
  - CREATE, WRITE, REMOVE, RENAME
  - LINK, SYMLINK
  - READIR, MKDIR, RMDIR
  - STATFS (get file system attributes)

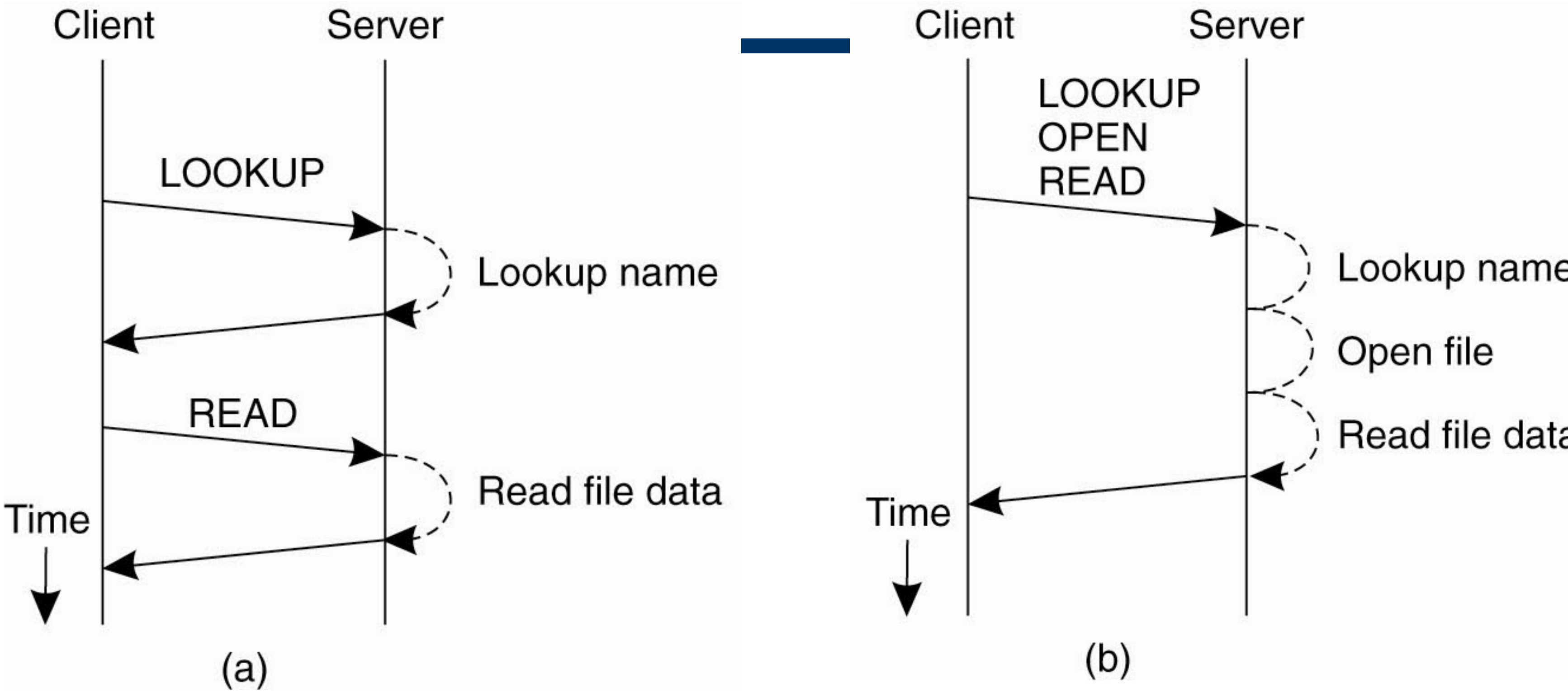
NFS V3 and V4 Operations

- V3 added:
  - READDIRPLUS, COMMIT (server cache!)
  - FSSTAT, FSINFO, PATHCONF
- V4 added:
  - COMPOUND (bundle operations)
  - LOCK (server becomes more stateful!)
  - PUTROOTFH, PUTPUBFH (no separate MOUNT)
  - Better security and authentication
  - Very different than V2/V3 → stateful

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

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?
- And if we cache... doesn’t that risk making things inconsistent?

Failures

- Server crashes
  - Data in memory but not disk lost
  - So... what if client does
    - seek() ; /* SERVER CRASH */; read()
    - If server maintains file position, this will fail (Why?). Ditto for open(), read()
- 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