CS 425 / ECE 428
Distributed Systems
Fall 2022

Indranil Gupta (Indy)

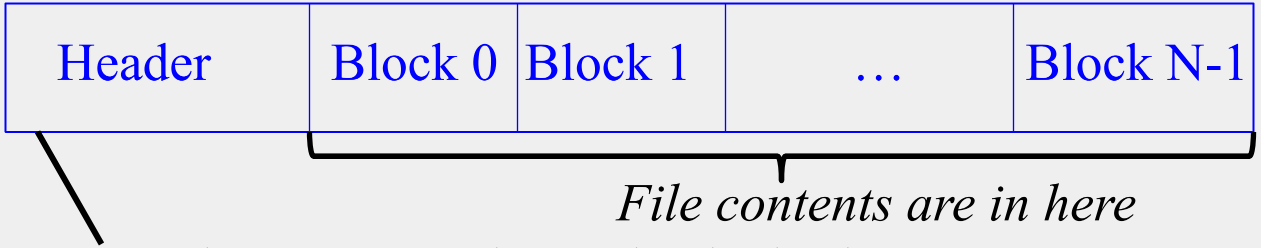
Lecture 23: Distributed File Systems

File System

- Contains files and directories (folders)
- Higher level of abstraction
  - Prevents users and processes from dealing with disk blocks and memory blocks

File Contents

- Typical File



  - Timestamps: creation, read, write, header
  - File type, e.g., .c, .java
  - Ownership, e.g., edison
  - Access Control List: who can access this file and in what mode
  - Reference Count: Number of directories containing this file
    - May be > 1 (hard linking of files)
    - When 0, can delete file

What about Directories?

- They’re just files!
- With their “data” containing
  - The meta-information about files the directory contains
  - Pointers (on disk) to those files

Unix File System: Opening and Closing Files

- Uses notion of *file descriptors*
  - Handle for a process to access a file
- Each process: Needs to open a file before reading/writing file
  - OS creates an internal datastructure for a file descriptor, returns handle
- *filedes*=open(*name, mode*)
  - mode = access mode, e.g., r, w, x
- *filedes*=creat(*name, mode*)
  - Create the file, return the file descriptor
- close(*filedes*)

Unix File System: Reading and Writing

- *error*=read(*filedes, buffer, num_bytes*)
  - File descriptor maintains a *read-write pointer* pointing to an offset within file
  - read() reads *num_bytes* starting from that pointer (into buffer), and *automatically advances pointer by num_bytes*
  - error returns the number of bytes read/written, or 0 if EOF, or -1 if error (errno is set)
- *error*=write(*filedes, buffer, num_bytes*)
  - Writes from buffer into file at position pointer
  - Automatically advances pointer by *num_bytes*
- *pos*=lseek(*filedes, offset, whence*)
  - Moves read-write pointer to position offset within file
  - *whence* says whether offset absolute or relative (relative to current pointer)

Unix File System: Control Operations

- *status*=link(*old_link, new_link*)
  - Creates a new link at second arg to the file at first arg
  - Old_link and new_link are Unix-style names, e.g., "/usr/edison/my_invention"
  - Increments reference count of file
  - Known as a "hard link"
    - Vs. "Symbolic/Soft linking" which creates another file pointing to this file; does not change reference count
- *status*=unlink(*old_link*)
  - Decrements reference count
  - If count=0, can delete file
- *status*=stat/fstat(*file_name, buffer*)
  - Get attributes (header) of file into *buffer*

Distributed File Systems (DFS)

- Files are stored on a server machine
  - client machine does RPCs to server to perform operations on file

Desirable Properties from a DFS

- Transparency: client accesses DFS files as if it were accessing local (say, Unix) files
  - Same API as local files, i.e., client code doesn't change
  - Need to make location, replication, etc. invisible to client
- Support concurrent clients
  - Multiple client processes reading/writing the file concurrently
- Replication: for fault-tolerance

Concurrent Accesses in DFS

- One-copy update semantics: when file is replicated, its contents, as visible to clients, are no different from when the file has exactly 1 replica
- At most once operation vs. At least once operation
  - Choose carefully
  - At most once, e.g., append operations cannot be repeated
  - *Idempotent* operations have no side effects when repeated: they can use at least once semantics, e.g., read at absolute position in file

Security in DFS

- Authentication
  - Verify that a given user is who they claim to be
- Authorization
  - After a user is authenticated, verify that the file they're trying to access is in fact allowed for that user
  - Two popular flavors
  - **Access Control Lists (ACLs)** = per file, list of allowed users and access allowed to each
  - **Capability Lists** = per user, list of files allowed to access and type of access allowed
    - Could split it up into capabilities, each for a different (user,file)

Let’s Build a DFS!

- We’ll call it our “Vanilla DFS”
- Vanilla DFS runs on a server, and at multiple clients
- Vanilla DFS consists of three types of processes
  - *Flat file service*: at server
  - *Directory service*: at server, talks to (i.e., “client of”) Flat file service
  - *Client service*: at client, talks to Directory service and Flat file service

Vanilla DFS: Flat File Service API

- Read(*file_id*, *buffer*, *position*, *num_bytes*)
  - Reads *num_bytes* from absolute *position* in file *file_id* into *buffer*
    - *File_id* is *not* a file descriptor, it's a unique id of that file
  - No automatic read-write pointer!
    - Why not? Need operation to be *idempotent* (at least once semantics)
  - No file descriptors!
    - Why not? Need servers to be *stateless*: easier to recover after failures (no state to restore!)
  - In contrast, Unix file system operations are neither idempotent nor stateless

Vanilla DFS: Flat File Service API (2)

- write(*file_id*, *buffer*, *position*, *num_bytes*)
  - Similar to read
- create/delete(*file_id*)
- get_attributes/set_attributes(*file_id*, *buffer*)

Vanilla DFS: Directory Service API

- *file_id* = lookup(*dir, file_name*)
  - file_id can then be used to access file via Flat file service
- add_name(*dir, file_name, buffer*)
  - Increments reference count
- un_name(*dir, file_name*)
  - Decrements reference count; if =0, can delete
- *list*=get_names(*dir, pattern*)
  - Like ls –al or dir, followed by grep or find

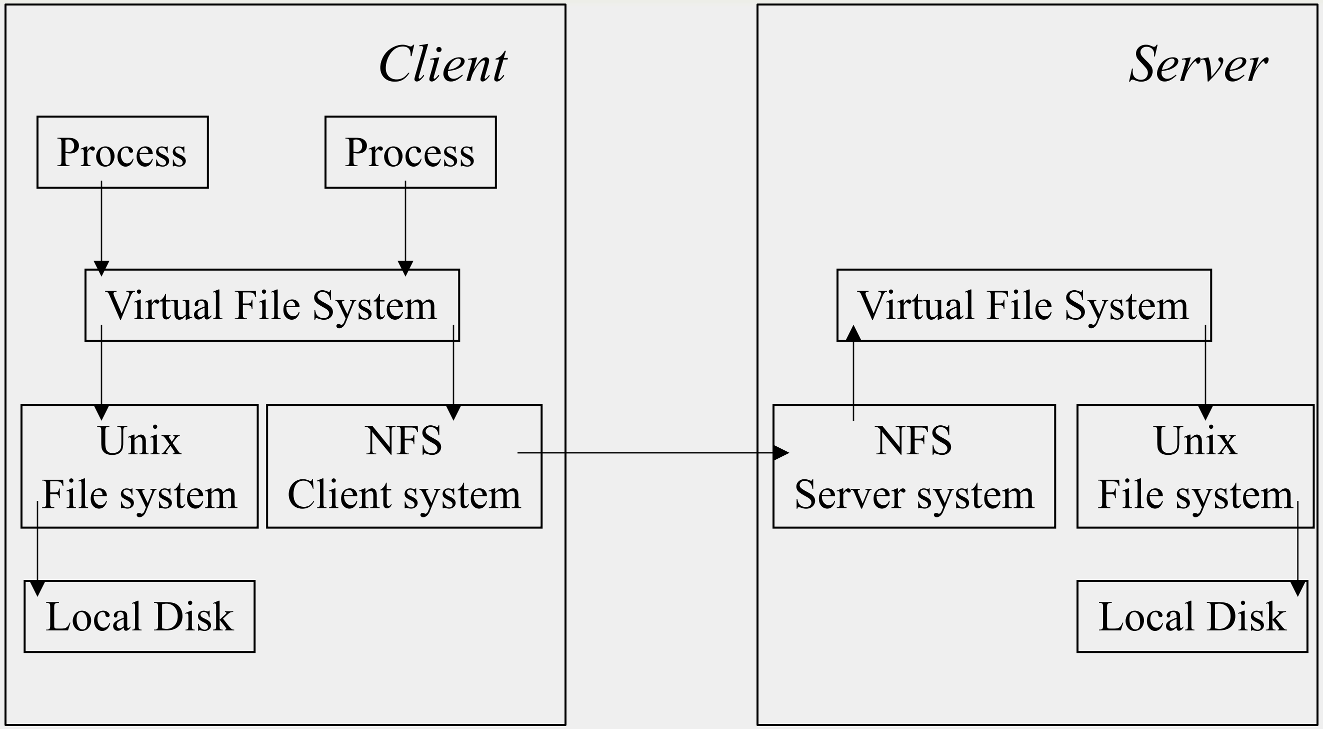
Can we Build a Real DFS Already?

- Next: Two popular distributed file systems
  - NFS and AFS

NFS

- Network File System
- Sun Microsystems, 1980s
- Used widely even today

NFS Architecture



NFS Client and Server Systems

- NFS Client system
  - Similar to our “Client service” in our Vanilla DFS
  - Integrated with kernel (OS)
  - Performs RPCs to NFS Server system for DFS operations
- NFS Server system
  - Plays the role of both Flat file service + Directory service from our Vanilla DFS
  - Allows *mounting* of files and directories
    - Mount /usr/tesla/inventions into /usr/edison/my_competitors
    - => Now, /usr/edison/my_competitors/foo refers to /usr/tesla/inventions/foo
    - Mount: Doesn’t clone (copy) files, just point to that directory now

Virtual File System Module

- Allows processes to access files via file descriptors
  - Just like local Unix files! So, local and remote files are indistinguishable (i.e., gives transparency)
  - For a given file access, decides whether to route to local file system or to NFS client system
- Names all files (local or remote) uniquely using “NFS file handles”
- Keeps a data structure for each mounted file system
- Keeps a data structure called v-node for all open files
  - If local file, v-node points to local disk block (called i-node)
  - If remote, v-node contains address of remote NFS server

Server Optimizations

- Server caching is one of the big reasons NFS is so fast with reads
  - Server Caching = Store, in memory, some of the recently-accessed blocks (of files and directories)
  - Most programs (written by humans) tend to have *locality of access*
    - Blocks accessed recently will be accessed soon in the future
- Writes: two flavors
  - Delayed write: write in memory, flush to disk every 30 s (e.g., via Unix sync operation)
    - Fast but not consistent
  - Write-through: Write to disk immediately before ack-ing client
    - Consistent but may be slow

Client Caching

- Client also caches recently-accessed blocks
- Each block in cache is tagged with
  - *Tc*: the time when the cache entry was last validated.
  - *Tm*: the time when the block was last modified at the server.
  - A cache entry at time *T* is valid if

    $(T - Tc < t)$ or $(Tm_{client} = Tm_{server})$.
  - *t=freshness interval*
    - Compromise between consistency and efficiency
    - Sun Solaris: *t* is set adaptively between 3-30 s for files, 30-60 s for directories
- When block is written, do a delayed-write to server

Andrew File System (AFS)

- Designed at CMU
  - Named after Andrew Carnegie and Andrew Mellon, the “C” and “M” in CMU
- In use today in some clusters (especially University clusters)

Interesting Design Decisions in AFS

- Two unusual design principles:
  - Whole file serving
    - Not in blocks
  - Whole file caching
    - Permanent cache, survives reboots

- Based on (validated) assumptions that
  - Most file accesses are by a single user
  - Most files are small
  - Even a client cache as “large” as 100MB is supportable (e.g., in RAM)
  - File reads are much more often that file writes, and typically sequential

AFS Details

- Clients system = *Venus* service
- Server system = *Vice* service
- Reads and writes are optimistic
  - Done on local copy of file at client (Venus)
  - When file closed, writes propagated to Vice
- When a client (Venus) opens a file, Vice:
  - Sends it entire file
  - Gives client a *callback promise*
- Callback promise
  - Promise that if another client modifies then closes the file, a callback will be sent from Vice to Venus
  - Callback state at Venus only binary: valid or canceled

Summary

- Distributed File systems
  - Widely used today
- Vanilla DFS
- NFS
- AFS
- Many other distributed file systems out there today!

Announcements

- HW4 and MP4 due after TG/Fall break
- MP3 due 11/6, demos following Monday.