NFS

ECE469, April 18

Yiying Zhang

Reading

- Comet Ch 48 (optionally 47 and 49)

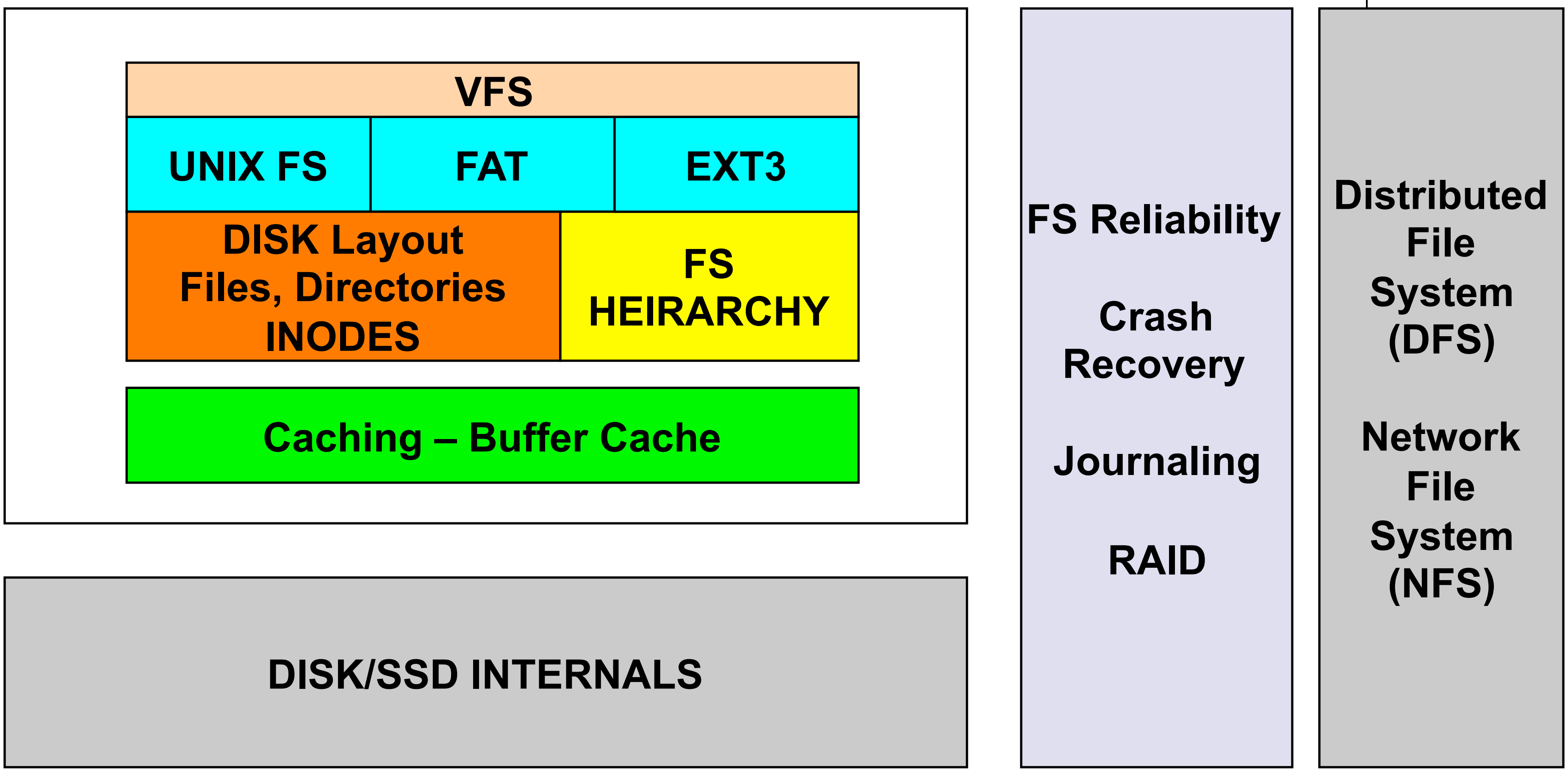
Misc

- ~~Rumor: “final will be much harder than midterm”~~

Lie!

- Today’s topic:
  - Advanced
  - But not hard: Demonstrate what great things we can build by putting concepts/techniques we learned in ECE469 together!

FS topics we have covered



DFS

- Definition: a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources

- Many DFS have been proposed and developed

Motivation

- Why are distributed file systems useful?
  - Access from multiple clients
    - Same user on different machines can access same files
  - Simplifies sharing
    - Different users on different machines can read/write to same files
  - Simplifies administration
    - One shared server to maintain (and backup)
  - Improve reliability
    - Add RAID storage to server

Challenges

- Transparent access
  - User sees single, global file system regardless of location
- Scalable performance
  - Performance does not degrade as more clients are added
- Fault Tolerance
  - Client and server identify and respond appropriately when other crashes
- Consistency
  - See same directory and file contents on different clients at same time
- Security
  - Secure communication and user authentication
- Tension across these goals
  - Example: Caching helps performance, but hurts consistency

NFS (Network File System)

- First commercially successful distributed file system:
  - Developed in 1984 by Sun Microsystems for their diskless workstations
  - Designed for robustness and "adequate performance"
  - Multiple versions (v2, v3, v4)
  - Widely used today

NFS Overview

- Remote Procedure Calls (RPC) for communication between client and server
- Client Implementation
  - Provides transparent access to NFS file system
    - UNIX contains Virtual File system layer (VFS)
    - Vnode: interface for procedures on an individual file
  - Translates vnode operations to NFS RPCs
- Server Implementation
  - Stateless: Must not have anything only in memory
  - Implication: All modified data written to stable storage before return control to client
    - Servers often add NVRAM to improve performance

NFS Design Objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***"Reasonable" performance***
  - Robustness and preservation of UNIX semantics were much more important

Naming properties

- *Location transparency:*
  - Name of the file does not reveal the file's physical storage location

- *Location independence:*
  - Name of file does not need to be changed when file's physical location changes

Two naming schemes

- Files named by combination of their host name and local name; guarantees a unique systemwide name
  - Neither location transparent
  - Nor location independent

- “Attach” remote directories to local directories, giving the appearance of a coherent directory tree, e.g. Sun’s NFS
- Use internal NFS operations to implement application APIs (POSIX)

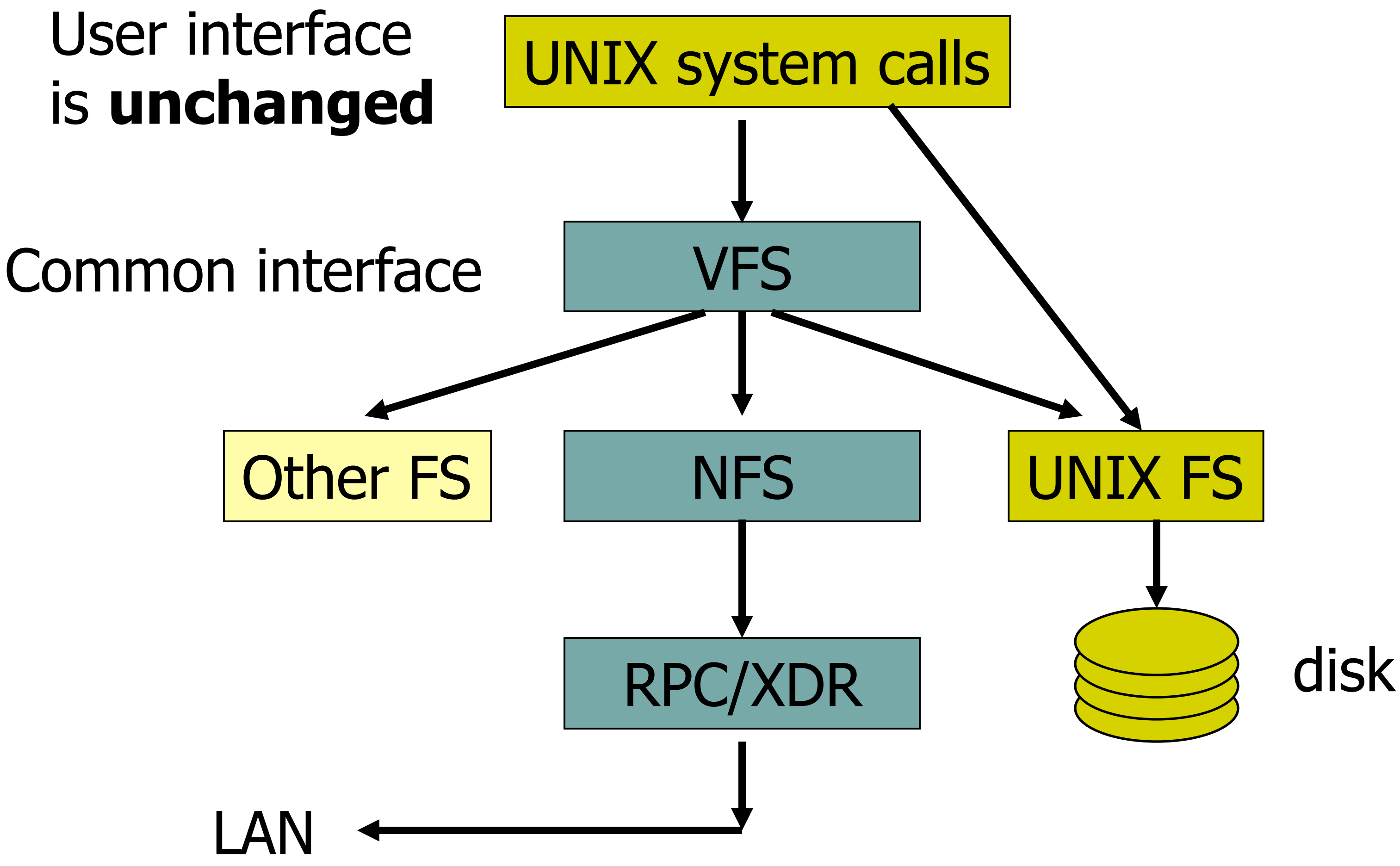
Implementation of transparency

- "*All computer science problems can be solved with an extra level of indirection*"

  -- David Wheeler

- What were the earlier manifestations of this in this class?

Client Side


User interface
is **unchanged**
UNIX system calls
Common interface
VFS
Other FS
NFS
UNIX FS
RPC/XDR
disk
LAN

Basic NFS Protocol

- Operations at NFS layer (applications do not execute these)
  - `lookup(dirfh, name) returns (fh, attributes)`
    - `Use mount protocol for root directory`
  - `create(dirfh, name, attr) returns (newfs, attr)`
  - `remove(dirfs, name) returns (status)`
  - `read(fh, offset, count) returns (attr, data)`
  - `write(fh, offset, count, data) returns attr`
  - `gettattr(fh) returns attr`

  - `What's missing here?`
    - `close No need to tell server: stateless server, more later`

- How to use these operations to implement file system system calls?

NFS Design Objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***"Reasonable" performance***
  - Robustness and preservation of UNIX semantics were much more important

NSF Key Ideas

- NSF key idea #1: Stateless server
  - Server not required to remember anything (in memory)
    - Which clients are connected, which files are open, …
  - Implication: All client requests have all the information to complete op
    - Example: Client specifies offset in file to write to
  - Why is this important for fast crash recovery?

- NSF Key idea #2: Idempotent server operations
  - Operation can be repeated with same result (no side effects)
  - Example: idempotent: a=b+1; Not idempotent: a=a+1;
  - Why is this important for crash recovery?

Advantages of stateless

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server
- Server state does not grow with more clients
- Simplifies the protocol
  - Client can always repeat any request

Consequences of stateless

- read and writes calls must specify offset
  - Server does not keep track of current position in the file
- But user will still use conventional UNIX APIs
- How should UNIX APIs be translated?
  - open() / close()
  - read() / write()

How to identify files in NFS?

- Can we still use inode?

- NFS use File handles

- **File handle** consists of
  - ***Filesystem id*** identifying disk partition
  - ***i-node number*** identifying file within partition
  - ***i-node generation number*** changed every time i-node is reused to store a new file

Filesystem id	i-node number	i-node generation number

Basic NFS Protocol

- Operations at NFS layer (applications do not execute these)
  - `lookup(dirfh, name) returns (fh, attributes)`
    - `Use mount protocol for root directory`
  - `create(dirfh, name, attr) returns (newfs, attr)`
  - `remove(dirfs, name) returns (status)`
  - `read(fh, offset, count) returns (attr, data)`
  - `write(fh, offset, count, data) returns attr`
  - `gettattr(fh) returns attr`

Remote lookup

- Returns a **file handle** instead of a file desc.
  - File handle specifies *unique location* of file

- **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**

Remote lookup

- To lookup **“/usr/joe/6360/list.txt”**

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)

Mapping UNIX System Calls to NFS Operations

- Unix system call: `fd = open("/dir/foo")`
  - Traverse pathname to get filehandle for foo
    - `dir_fh = lookup(root_dir_fh, "dir");`
    - `fh = lookup(dir_fh, "foo");`
  - Record mapping from `fd` file descriptor to `fh` NFS filehandle
  - Set initial file offset to 0 for `fd`
  - Return `fd` file descriptor
- Unix system call: `read(fd,buffer,bytes)`
  - Get current file `offset` for `fd`
  - Map `fd` to `fh` NFS filehandle
  - Call `data = read(fh, offset, bytes)` and copy `data` into `buffer`
  - Increment file `offset` by `bytes`
- Unix system call: `close(fd)`
  - Free resources assocatiated with fd
  - No need to tell server: stateless server

NFS Design Objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***"Reasonable" performance***
  - Robustness and preservation of UNIX semantics were much more important

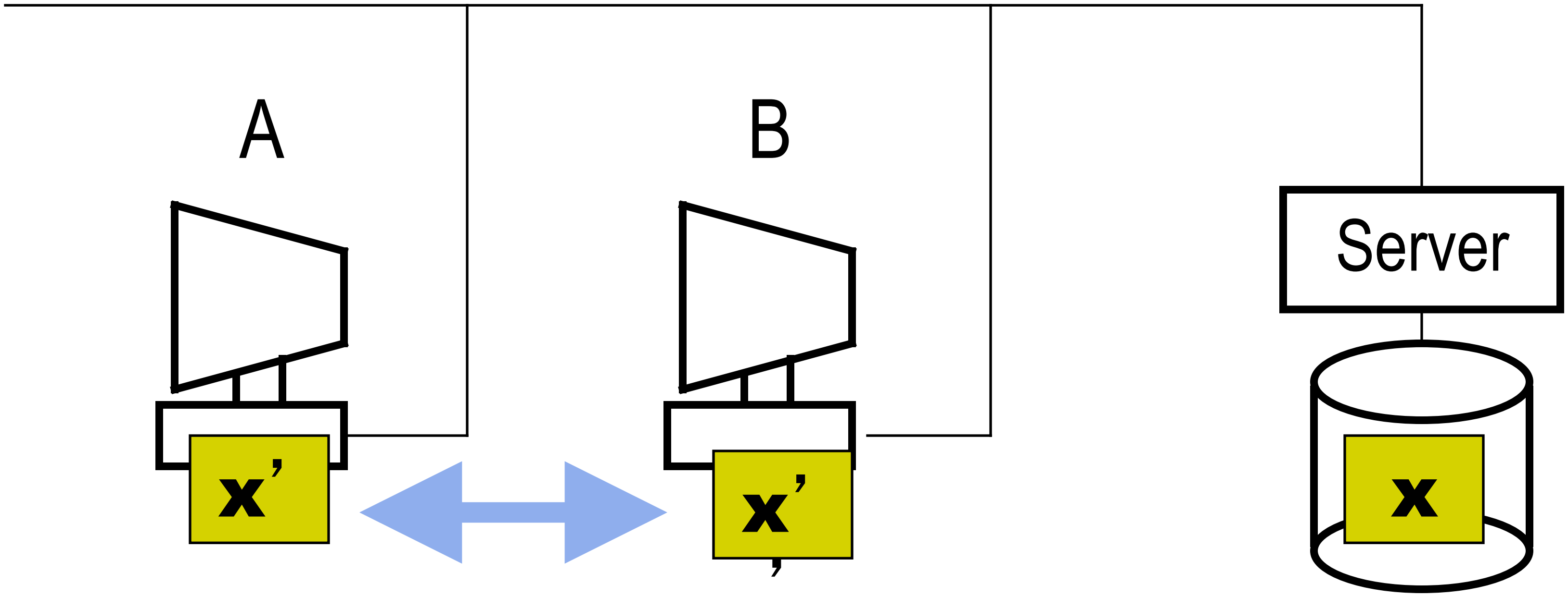
Client-side Caching

- Caching needed to improve performance
  - Reads: Check local cache before going to server
  - Writes: Only periodically write-back data to server
  - Why avoid contacting server
    - Avoid slow communication over network
    - Server becomes scalability bottleneck with more clients
- Two types of client caches
  - data blocks
  - attributes (metadata)

Cache Consistency

- Problem: Consistency across multiple copies (server and multiple clients)
  - How to keep data consistent between client and server?
    - If file is changed on server, will client see update?
    - Determining factor: Read policy on clients
  - How to keep data consistent across clients?
    - If write file on client A and read on client B, will B see update?
    - Determining factor: Write and read policy on clients

Cache Consistency Problem



Inconsistent updates

NFS Consistency: Reads

- Reads: How does client keep current with server state?
  - Attribute cache: Used to determine when file changes
    - File open: Client checks server to see if attributes have changed
      - If haven't checked in past T seconds (configurable, T=3)
    - Discard entries every N seconds (configurable, N=60)
  - Data cache
    - Discard all blocks of file if attributes show file has been modified
- Eg: Client cache has file A's attributes and blocks 1, 2, 3
  - Client opens A:
  - Client reads block 1 => cache
  - Client waits 70 seconds
  - Client reads block 2 => cache
  - Block 3 is changed on server
  - Client reads block 3 => cache, get old value
  - Client reads block 4 => fetch from server
  - Client waits 70 seconds
  - Client reads block 1 => fetch from server

NFS Consistency: Writes

- Writes: How does client update server?
  - Files
    - Write-back from client cache to server every 30 seconds
    - Also, Flush (write all dirty data) on close() (AKA *flush-on-close*)
  - Directories
    - Synchronously write to server (write through)

- Example: Client X and Y have file A (blocks 1,2,3) cached
  - Clients X and Y open file A
  - Client X writes to blocks 1 and 2
  - Client Y reads block 1 => cache
  - 30 seconds later...
  - Client Y reads block 2 => cache, get old value
  - 40 seconds later...
  - Client Y reads block 1 => server

Conclusions

- Distributed file systems
  - Important for data sharing
  - Challenges: Fault tolerance, scalable performance, and consistency
- NFS: Popular distributed file system
  - Key features:
    - Stateless server, idempotent operations: Simplifies fault tolerance
    - Crashed server appears as slow server to clients
  - Client caches needed for scalable performance
    - Rules for invalidating cache entries and flushing data to server are not straight-forward
    - Data consistency very hard to reason about