Announcements

- P0: Grades released on Gradescope
- For any private communication, use course staff email < ds-staff-f21-private@lists.andrew.cmu.edu>. Not individual instructor email addresses.
- For everyone's safety:
  - Please do not congregate after the class for Q/A -- ask questions during the lecture or make use of Piazza and office hours
  - If you are sick, please watch the lectures remotely
  - Wear your mask properly **covering your nose and mouth entirely at all times during the lecture**

15-440/640 Distributed Systems

Distributed File Systems

(… continued)

Quick recap

- 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

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 are DFSs Useful?

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

Challenges

- Heterogeneity (lots of different computers & users)
- Scale (10s of thousands of users!)
- Security (my files! hands off!)
- Failures
- Concurrency

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)
- If you change the workload assumptions the design parameters change!

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 → requires kernel support

- **Communication layer:**
  - How are requests sent to server?

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

A Simple Approach

- Use RPC to forward every filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.

Needing to hit the server for every detail **impairs performance and scalability.**

Caching

- **Client-side caching**: Having a copy of the data on client machine

- Consistency challenges

- Read-only files → easy
- Data that is written by other machines ("**cache staleness**")→
  - How to know that the data has changed?
- Data written by the client machine ("**update visibility**")→
  - When is data written to the server?

Approaches to handle cache staleness

- Update propagation policy

- Key ideas
  - Broadcast invalidations
  - Check on use
  - Callbacks
  - Leases (we will continue looking into this idea today)

4. Leases

- Granting exclusive/shared control of the cached objects for a ***limited amount of time***
- **Lease period**: duration of the control
- **Lease renewal**: control expires without renewal
  - Client has to request lease renewal
  - Need to take latency of communication into account

- Read lease vs write lease
  - multiple sites can obtain read lease
  - only one can get write lease

- Clock synchronization
  - Not necessary: absolute time not relevant
  - But clock tick rate matters

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. Ditto for open(), read()
- Lost messages: what if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created a new file called foo?
- Client crashes
  - Might lose data in client cache

Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
- Generally, clients do not cache data on local disks

- 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

NFS's Failure Handling – Stateless Server

- 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

- Stateless protocol: requests specify exact state.

  read() → read([position])

  no seek on server

NFS's Failure Handling – Stateless Server

- Operations are **idempotent**
- How can we ensure this?
- Lost messages: what if we lose acknowledgement for delete("foo"). And in the meantime, another client created a new file called foo?
  - Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.
- Not perfect → e.g., mkdir

Client Caching in AFS

- NFS gets us partway there, but
  - Not ideal for handling large scale (*, - you can buy huge NFS appliances today that will, but they're $$$-y).
  - Is very sensitive to network latency

- How can we improve this?
  - More aggressive caching (AFS caches on disk in addition to just in memory)
  - Prefetching (on open, AFS gets entire file from server, making later ops local & fast).
    - Remember: with traditional hard drives, large sequential reads are much faster than small random reads. So easier to support (client a: read whole file; client B: read whole file) than having them alternate. Improves scalability, particularly if client is going to read whole file anyway eventually.

Client Caching in AFS

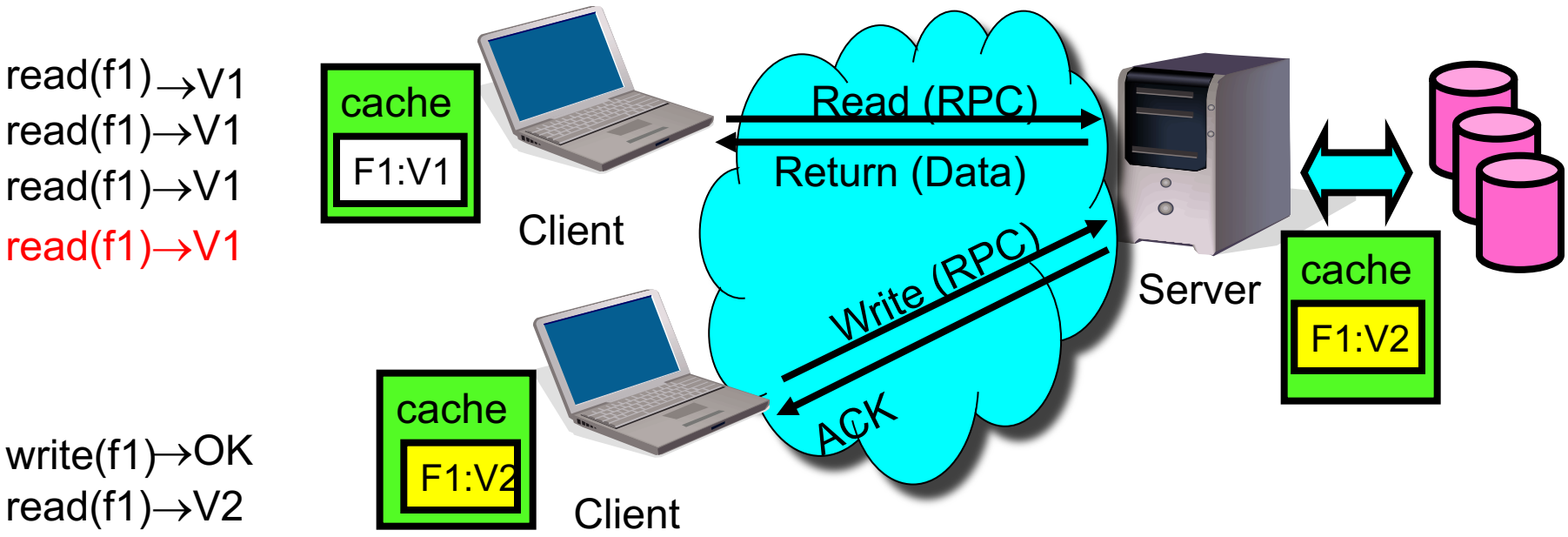
- Callbacks!

- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone “who has which files cached?”
- What if client crashes?
  - Must revalidate any cached content it uses since it may have missed callback

AFS v2 RPC Procedures

- Some procedures that are not in NFS related to callbacks
  - Fetch: return status and optionally data of a file or directory, and *place a callback on it*
  - RemoveCallBack: specify a file that the client has flushed from the local machine
  - BreakCallBack: from server to client, revoke the callback on a file or directory
    - What should the client do if a callback is revoked?

Update/consistency semantics



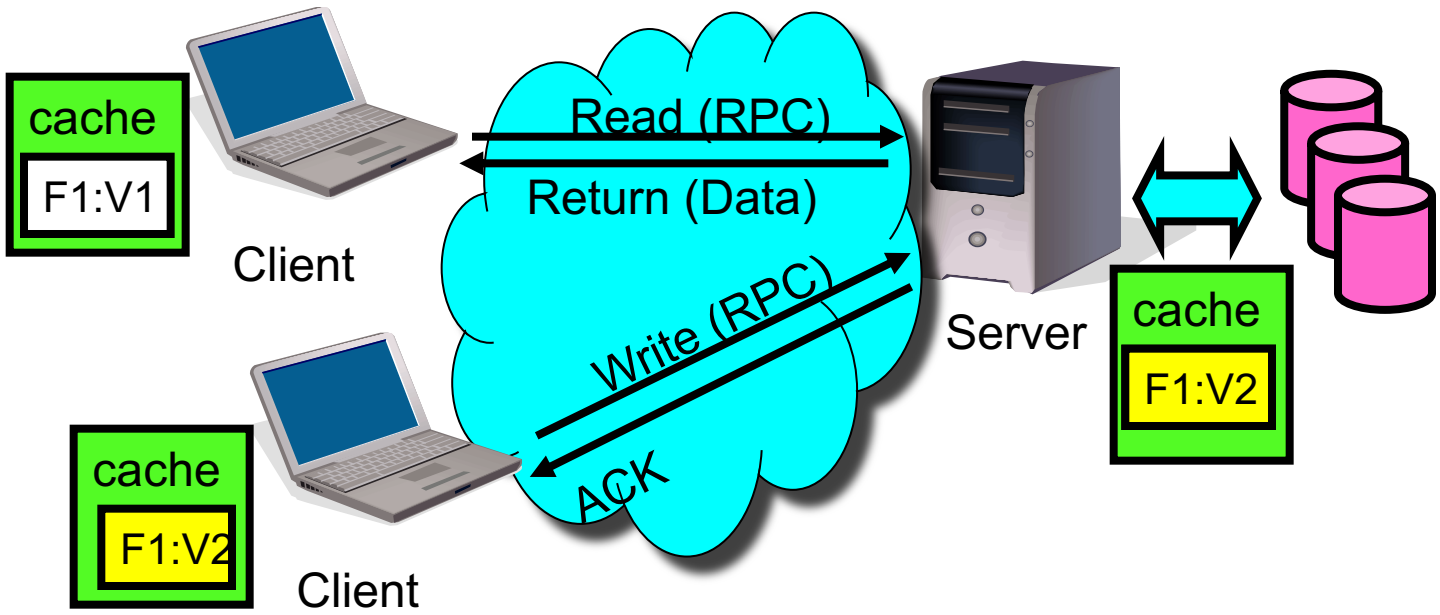
Ideal (One copy semantics)

- As though there is a single copy of the file that everyone is editing
  - No functional differences (as though no caches)
  - Difficult to achieve in distributed setting
    - Network (connection) failures
    - Lots of read/write sharing across clients

One copy semantics

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V2

write(f1)→OK
read(f1)→V2



File Access Consistency

- In UNIX local file system, concurrent file reads and writes have “sequential” consistency semantics
  - Each file read/write from user-level app is an atomic operation
    - The kernel locks the file vnode
  - Each file write is immediately visible to all file readers
- Neither NFS nor AFS provides such concurrency control
  - NFS: “sometime within 30 seconds”
  - AFS: session semantics for consistency

NFS Write Policy

- Dirty data buffered on the client machine until file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost

- 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 Results

- NFS provides transparent, remote file access

Advantages:

- Simple, portable, *popular* (it's gotten a little more complex over time, but...)
- No network traffic if open/read/write/close can be done locally.

Disadvantages:

- Weak consistency semantics
  - Simply unacceptable for some distributed applications
  - Productivity apps tend to tolerate such loose consistency
- Requires hefty server resources to scale (write-through, server queried for lots of operations)

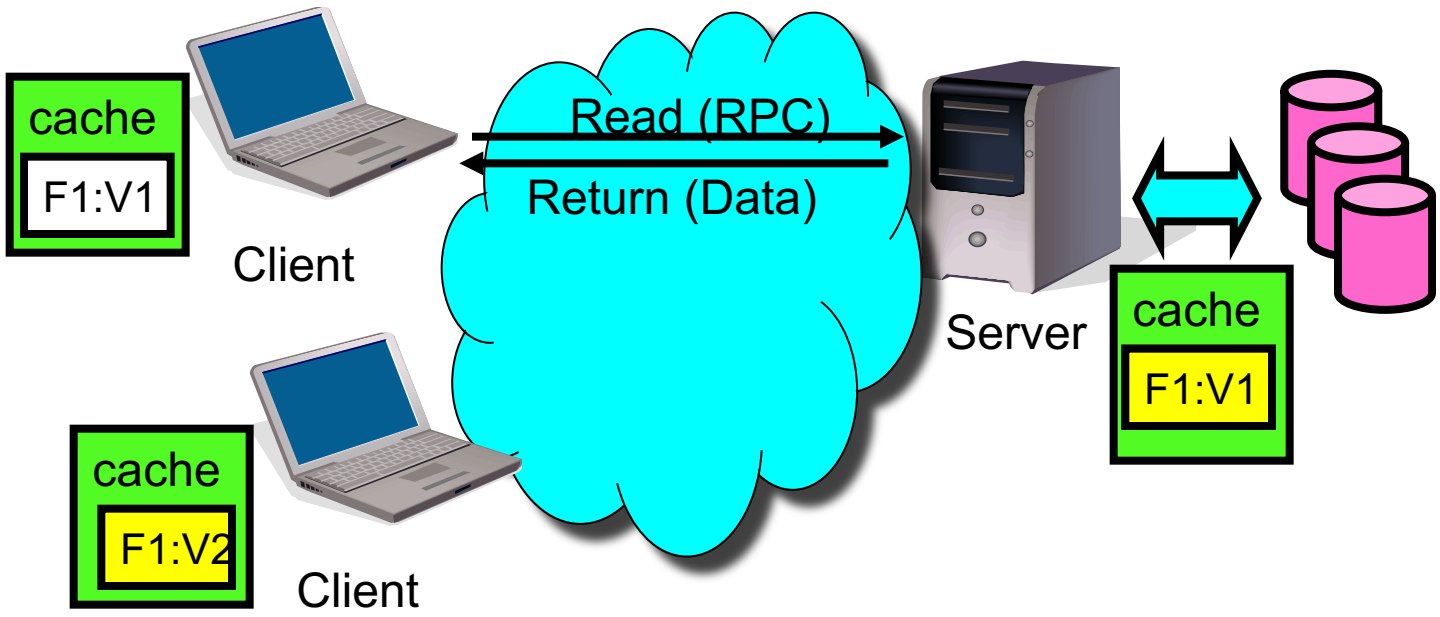
Session Semantics in AFS v2

- What it means:
  - A file write is visible to processes on the same machine immediately, but not visible to processes on other machines until the file is closed
  - When a file is closed, changes are visible to new opens, but are not visible to “old” opens
  - Last writer (i.e. the last close()) wins

- Implementation
  - Dirty data are buffered at the client machine until file close, then flushed back to server, which leads the server to send “break callback” to other clients

Session semantics

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

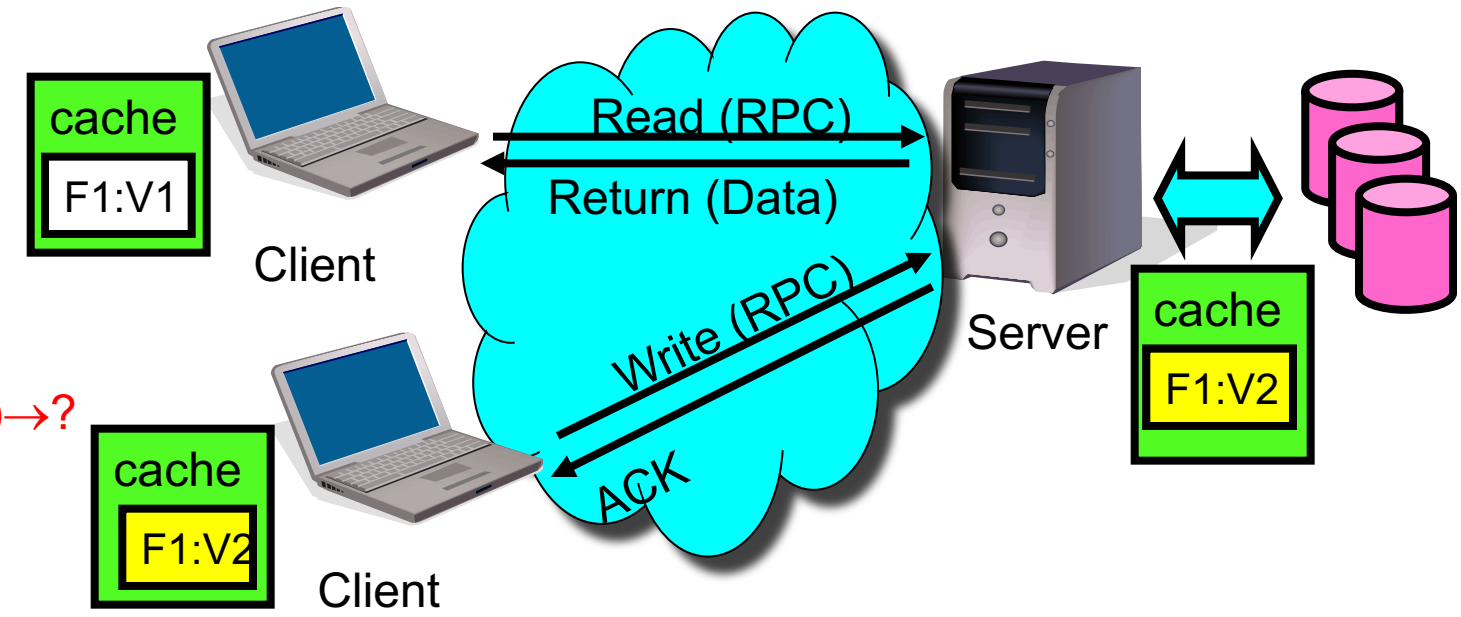


write(f1)→OK
read(f1)→V2

Session semantics

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V1
close(f1)
Open & read(f1)→?

write(f1)→OK
read(f1)→V2
close(f1)



AFS: Cache Consistency Timeline

Client$_1$			Client$_2$		Server	*Comments*
P_1	P_2	Cache	P_3	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P_3
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

Figure 50.3: **Cache Consistency Timeline**

Source:
Remzi, OS Book, "AFS"

AFS Write Policy

- Writeback cache (specifically “session semantics”)
  - Opposite of NFS “every write is sacred”
  - Store back to server
    - When cache overflows
    - On close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts “assumed rare”
  - Who wants to see a half-written file? (matters for collaborative editing applications though!)
- AFS also operates on a file granularity, so the last writer wins

AFS vs NFS

- AFS has lower server load than NFS
  - More files cached on clients
  - Callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is much slower than from another machine's memory over LAN
- For both:
  - Central server is bottleneck: all reads and writes hit it at least once;
  - is a single point of failure.
  - is costly to make them fast, beefy, and reliable servers.

Outline

- Why Distributed File Systems?

- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples

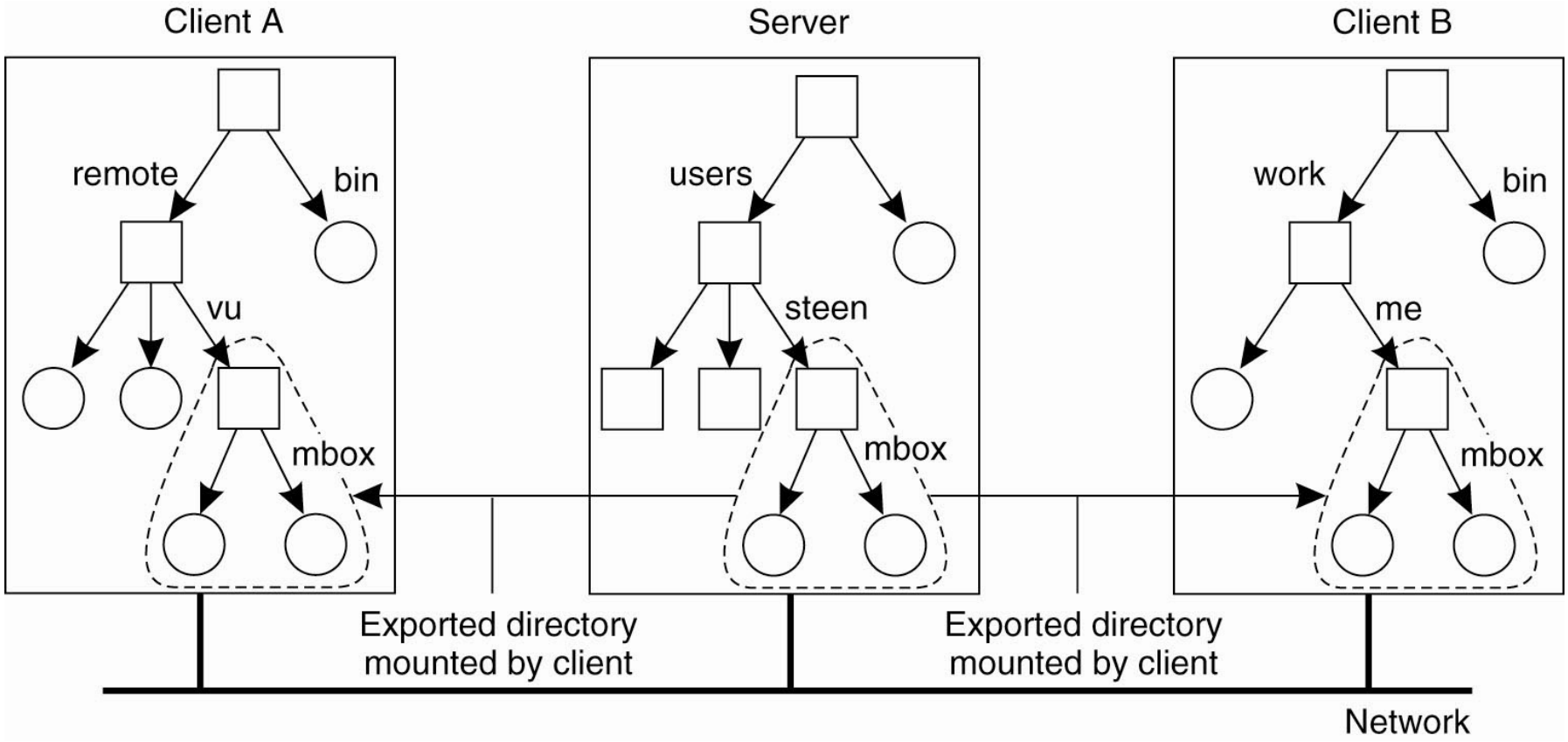
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Security

Naming

Naming in NFS

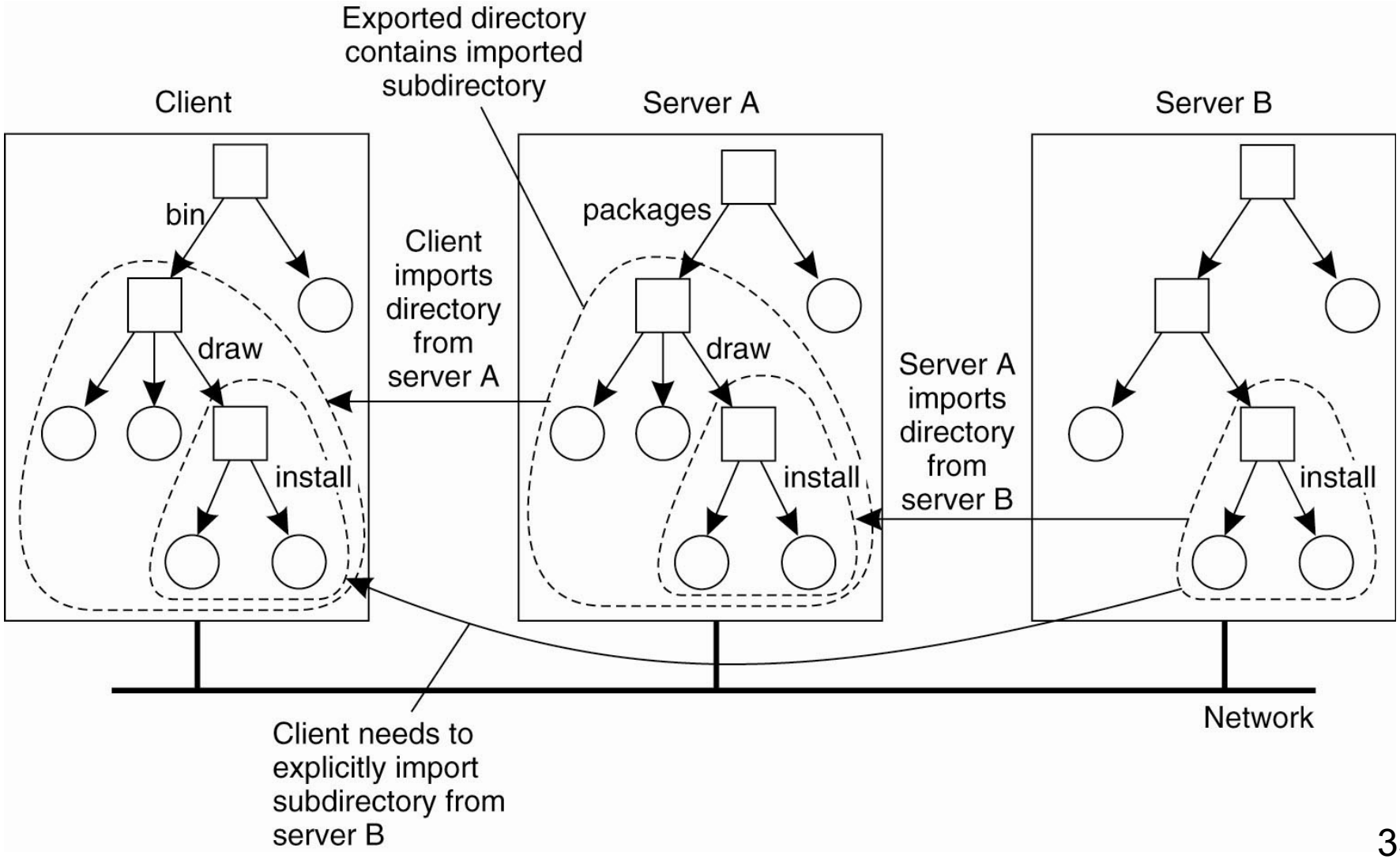
- NFS: clients mount NFS volume where they want
  - Server: export /root/disk1
  - Client1: mount server:/root/disk1 /remote
  - Client2: mount server:/root/disk1 /home/rashmi

Naming in NFS (1)



No naming transparency since both clients have the files (eg.mbox) stored in different hierarchcal namespaces.

Naming in NFS (2)



Naming

- NFS: clients mount NFS volume where they want
  - Server: export /root/disk1
  - Client1: mount server:/root/disk1 /remote
  - Client2: mount server:/root/disk1 /home/yuvraj
- AFS: name space consistent across clients
  - Global name space
    - Global directory /afs;
    - Client1: /afs/andrew.cmu.edu/disk1/
    - Client2: /afs/andrew.cmu.edu/disk1/
  - Each file is identified as fid = <vol_id, vnode #, unique identifier>
  - All AFS servers keep a copy of "**volume location database**", which is a table of vol_id→ server_ip mappings
- More details in the textbook (specified in additional reading)

Implications on Location Transparency

- NFS: no transparency
  - If a directory is moved from one server to another, client must remount

- AFS: transparency
  - If a volume is moved from one server to another, only the volume location database on the servers needs to be updated

Outline

- Why Distributed File Systems?

- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples

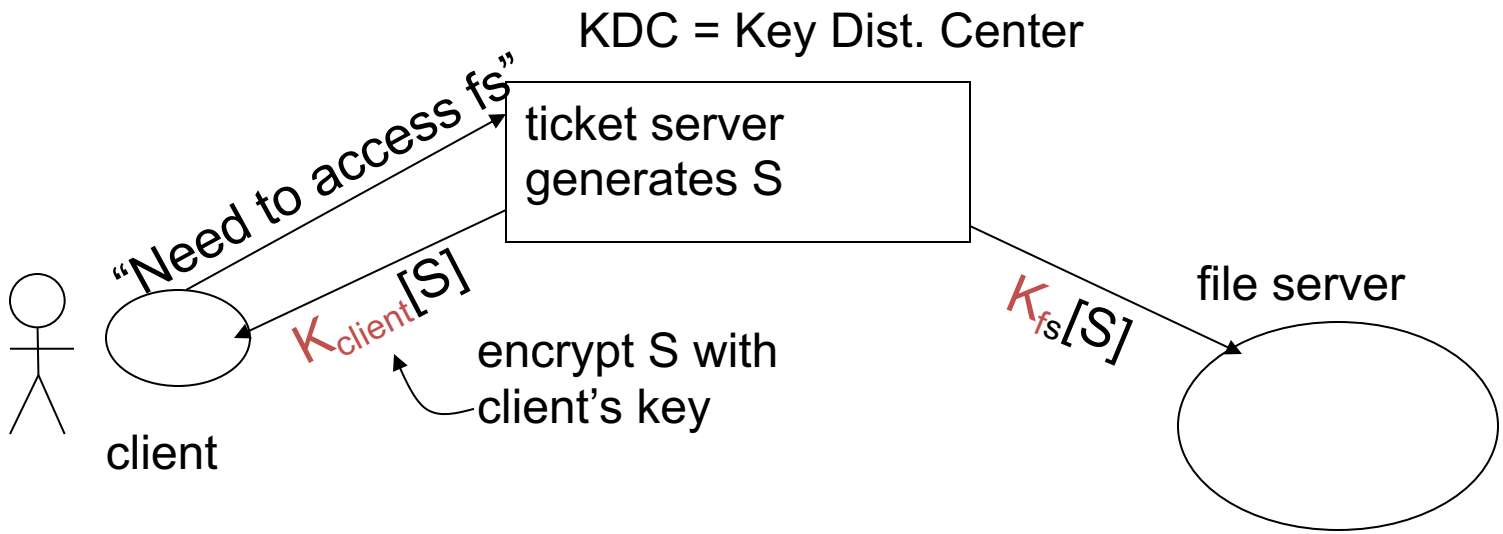
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Security

User Authentication and Access Control

- User X logs onto workstation A, wants to access files on server B
  - How does A tell B who X is?
  - Should B believe A?

Widely used solution: Kerberos

- Based on symmetric key cryptography (shared secrets)
  - User proves to KDC who he is; KDC generates shared secret between client and file server



S: specific to {client,fs} pair;
"short-term session-key"; expiration time (e.g. 8 hours)

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Security
- Disconnected operation with CODA as example
  - Optimistic and pessimistic replica control

Background on CODA

- We work in many different places
- Mobile Users appeared in 1990s
  - 1st Thinkpad, cell phones, ..
- Network is slow and not stable, clients ”powerful”
- We work at client without network connectivity

CODA

- Successor of the very successful Andrew File System (AFS)
- Allows owners of laptops to operate them in ***disconnected mode (Opposite of ubiquitous connectivity)***
- Developed as a research project at CMU!

Accessibility

- Must handle two types of failures
  - ***Server failures:***
    - Data servers are ***replicated***
  - ***Communication failures*** and ***voluntary disconnections***

Replica Control

- Pessimistic
  - Disable all partitioned writes
  - Require a client to acquire control (lock) of a cached object ***prior*** to disconnection

- Optimistic
  - Assuming no others touching the file
  - conflict detection

  \+ fact: low write-sharing

  \+ high availability: access anything in range

Pessimistic Replica Control

- ***Pessimistic replication control protocols*** guarantee the consistency of replicated in the presence of ***any non-Byzantine failures***
  - Typically require a **quorum** of replicas to allow access to the replicated data
  - Would **not** support disconnected mode
  - We shall cover Byzantine Faults and Failures later.

Pessimistic Replica Control

- Would require client to acquire ***exclusive*** (RW) or ***shared*** (R) control of cached objects before accessing them in disconnected mode:
  - Acceptable solution for voluntary disconnections
  - Does not work for involuntary disconnections
- What if the laptop remains disconnected for a long time?

Leases?

- We could grant exclusive/shared control of the cached objects for a ***limited amount of time***

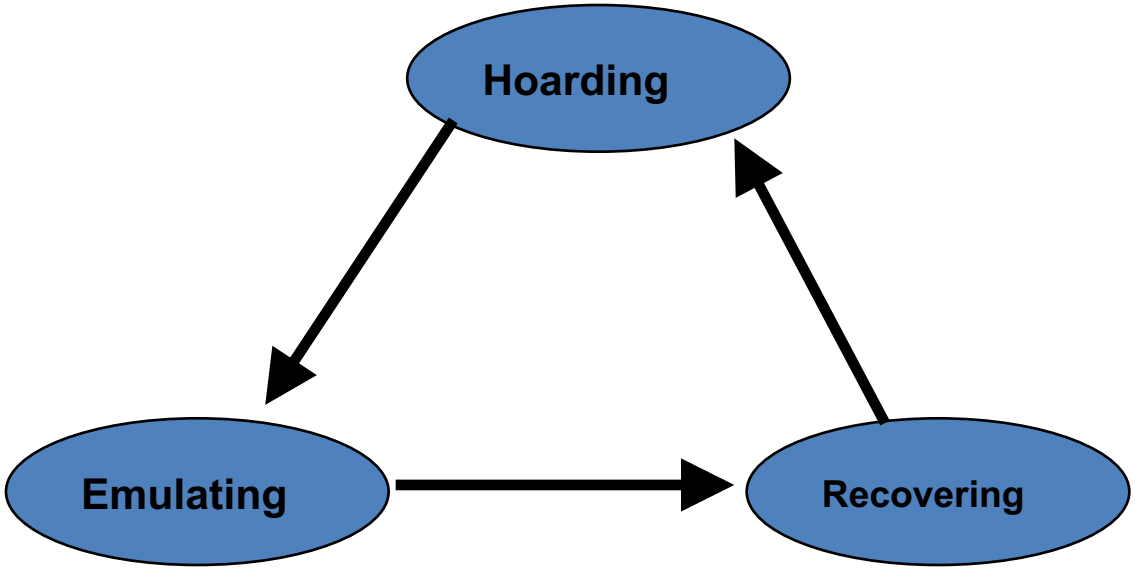
- Works very well in ***connected mode***
  - Reduces server workload

- Would only work for very **short disconnection periods**

Optimistic Replica Control

- ***Optimistic replica control*** allows access in ***every*** disconnected mode
  - Tolerates temporary inconsistencies
  - Promises to detect them later
  - Provides ***much higher data availability***

Coda States



1. ***Hoarding:***
   Normal operation mode
2. ***Emulating:***
   Disconnected operation mode
3. ***Reintegrating:***
   Propagates changes and detects inconsistencies

Hoarding

- Hoard useful data for disconnection
- Balance the needs of connected and disconnected operation
  - **Cache size is limited**
  - Unpredictable disconnections
- Uses user specified preferences + usage patterns to decide on files to keep in hoard

Emulation

- In emulation mode:
  - Attempts to access files that are not in the client caches appear as failures to application
  - All changes are written in a persistent log, the client modification log (CML)
  - Coda removes from log all obsolete entries like those pertaining to files that have been deleted

Reintegration

- When workstation gets reconnected, Coda initiates a ***reintegration process***
  - Performed one volume at a time
  - Ships replay log to all volumes
  - Each volume performs a log replay algorithm

- Only care about write/write conflict
  - Conflict resolution succeeds?
    - Yes. Free logs, keep going…
    - No. Save logs to a tar. Ask for help

- In practice:
  - No Conflict at all! Why?
  - Over 99% modification by the same person
  - Two users modify the same object within a day: <0.75%

Outline

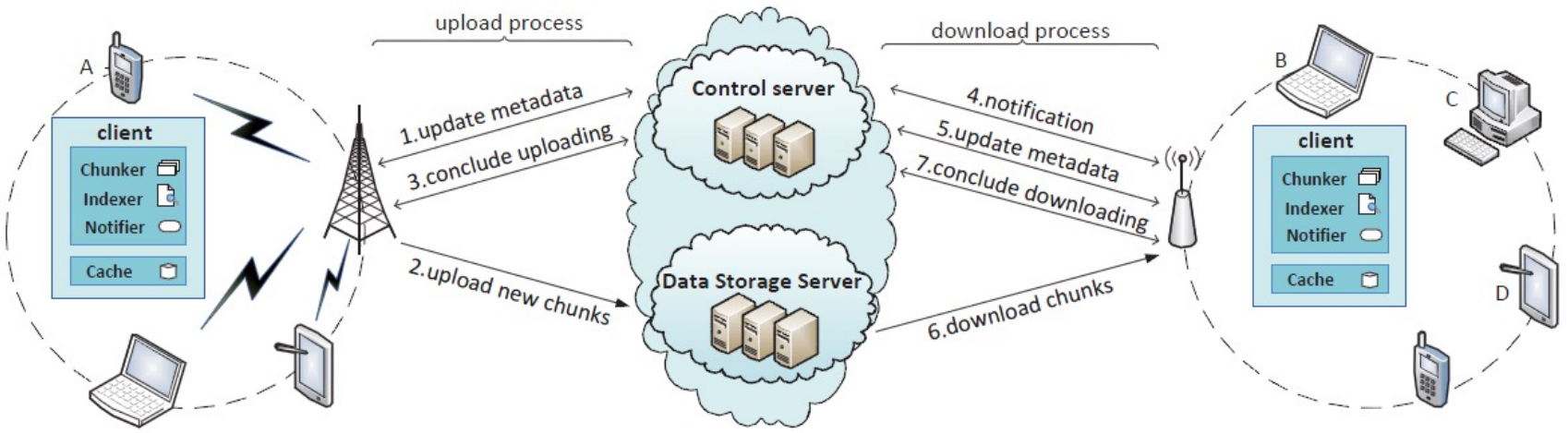
- Why Distributed File Systems?

- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples

- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Security
- Disconnected operation with CODA as example
- Other popular DFSs: Dropbox, G drive, etc.

Other popular DFSs

- Dropbox, Google Drive, OneDrive, BOX
- 100s of Millions of users, syncing petabytes (?)
- Basic function: Storing, sharing, synchronizing data between multiple devices, anytime, over any network
- General architecture (esp. Dropbox)



Picture Credit: Yong Cui, QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services

Features and Comparisons

- Chunking: splitting a large file into multiple data units
- Bundling: multiple small chunks as a single chunk
- Deduplication: avoiding sending existing content in the cloud
- Delta-encoding: transmit only the modified portion of a file

Capabilities	Windows			
	Dropbox	Google Drive	OneDrive	Seafile
Chunking	4 MB	8 MB	var.	var.
Bundling	√	×	×	×
Deduplication	√	×	×	√
Delta encoding	√	×	×	×
Data compression	√	√	×	×

Question: Dropbox's consistency model for conflicts?

Question: Why don't we do data deduplication always?

Summary

- Distributed filesystems almost always involve a tradeoff: consistency, performance, scalability.
- Client-side caching is a fundamental technique to improve scalability and performance
  - But raises important questions of cache consistency
- We'll see a related tradeoffs, also involving consistency, in a while: the CAP tradeoff. Consistency, Availability, Partition-resilience.