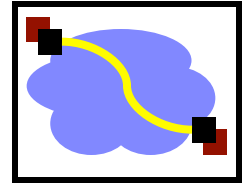


# 416 Distributed Systems

Distributed File Systems 3

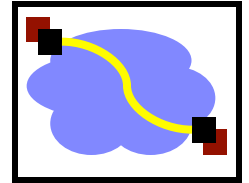
Jan 20, 2017

# Outline



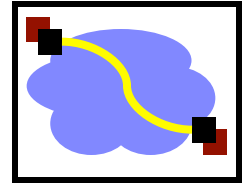
- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - **Caching**
  - Consistency
  - Naming
  - Authentication and Access Control

# Client Caching in NFS v2



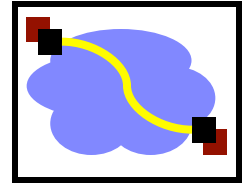
- Cache both **clean and dirty file data** and file attributes
  - Memory cache
  - Sub-file caching granularity
- File attributes (e.g., last modified time) in the cache expire after 60 seconds (file data doesn't expire)
  - Will retrieve updated attributes from server every 60s
- If server has a more recent modified time, grab the up-to-date data in cache from server
- Dirty data are buffered (in cache) on the client until file close or up to 30 seconds
  - If the machine crashes before then, the changes are lost

# Let's look back at NFS



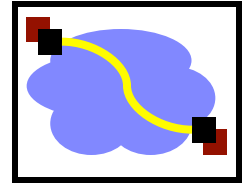
- NFS gets us partway there, but
  - Probably doesn't handle scale (\* - you can buy huge NFS appliances today that will, but they're \$\$\$-y).
  - Is very sensitive to network latency
- How can we improve this?
  - **More aggressive caching** (AFS caches on disk in addition to just in memory)
  - **Prefetching** (on open, AFS gets entire file from server, making later ops local & fast).

# Client Caching in AFS



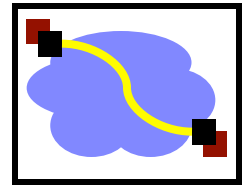
- Callbacks! Clients register with server that they have a copy of file;
  - Server tells them (calls them back): “Invalidate” if the file changed (but only does so on file close!)
  - This trades state for improved consistency
- Key AFS bit: read from local disk copy unless server indicates new copy exists (via callback)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from clients
    - ask everyone “who has which files cached?”

# AFS v2 RPC Procedures



- Procedures that are not in NFS
  - Fetch: from client to server, return status and optionally data of (entire) file or directory, and add a callback on it
  - RemoveCallBack: from C to S, specify a file that the client has flushed from the local machine
  - BreakCallBack: from S to C, revoke the callback on a file or directory (this is the callback call to client)
    - What should the client do if a callback is revoked?
      - Delete existing cached copy / refetch from server on open
  - Store: from S to C, store the status and optionally data of a file

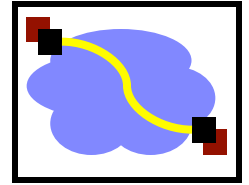
# Session semantics in AFS



- P1 and P2 local to Client1
- Clients 1/2 concurrent

Client <sub>1</sub>			Client <sub>2</sub>		Server	Comments
P <sub>1</sub>	P <sub>2</sub>	Cache	P <sub>3</sub>	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 see writes immediately
	read() → B	B		-	A	
	close()	B		-	A	
		B	open(F)	A	A	Remote processes do not see writes...
		B	read() → A	A	A	
		B	close()	A	A	
close()		B		<del>A</del>	B	... until close() has taken place
		B	open(F)	B	B	
		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		<del>C</del>	D	Unfortunately for P <sub>3</sub> the last writer wins
		D	open(F)	D	D	
		D	read() → D	D	D	
		D	close()	D	D	

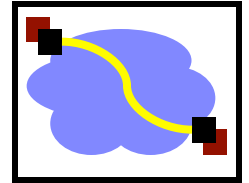
# Outline



- 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

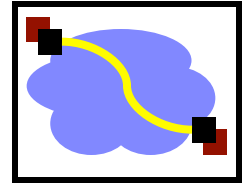


# Topic 2: File Access Consistency



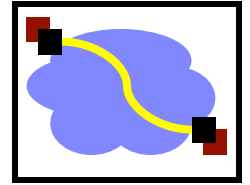
- In UNIX local file system, concurrent file reads and writes have “sequential” consistency semantics
  - Each file read/write from user-level app is an atomic operation
    - The kernel locks the file vnode
  - Each file write is immediately visible to all file readers
- Neither NFS nor AFS provides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

# Session Semantics in AFS v2



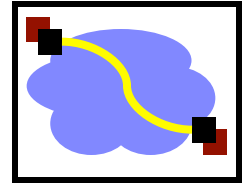
- What it means:
  - A file write is visible to processes on the same box immediately, but not visible to processes on other machines until the file is closed
  - When a file is closed, changes are visible to new opens, but are not visible to “old” opens
    - **Last closer wins!**
  - All other file operations are visible everywhere immediately
- 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

# AFS Write Policy



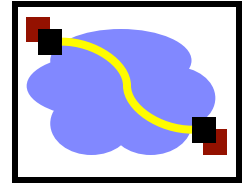
- Writeback cache (in contrast with write through)
  - Opposite of NFS “every write is sacred”
  - Store chunk back to server
    - When cache overflows
    - On last user close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts “assumed rare”
  - Who wants to see a half-written file?

# Results for AFS



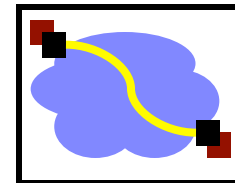
- Lower server load than NFS
  - More files cached on clients
  - Callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is **much** slower than from another machine's memory over LAN (better with SSD: ~1ms to read 1MB)
- For both:
  - Central server 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



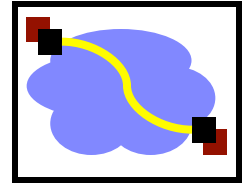
- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

# Topic 3: Name-Space Construction and Organization



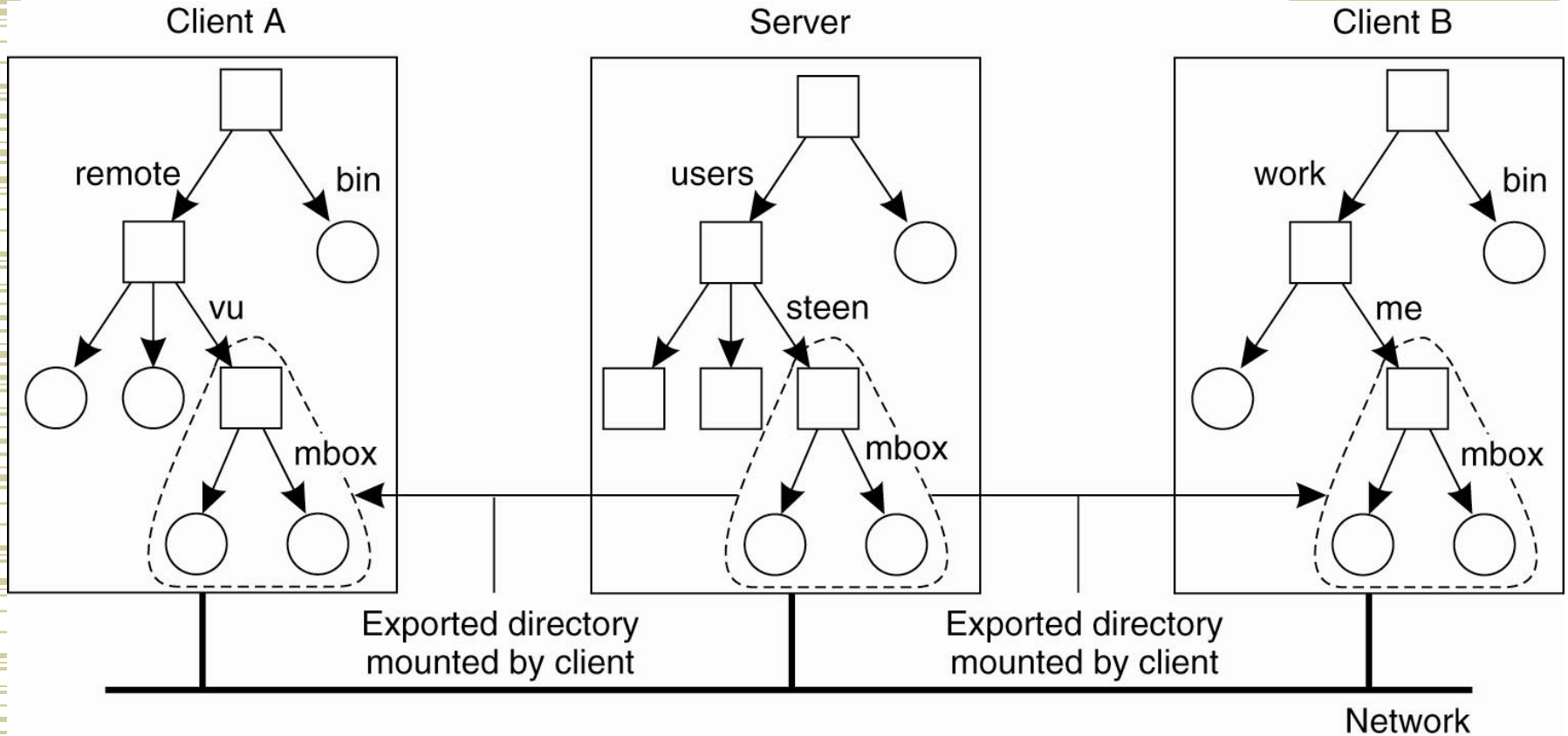
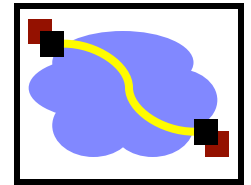
- NFS: per-client linkage
  - Server: export /root/fs1/
  - Client: mount server:/root/fs1 /fs1
- AFS: global name space
  - Name space is organized into Volumes
    - Global directory /afs;
    - /afs/cs.wisc.edu/vol1/...; /afs/cs.stanford.edu/vol1/...
  - Each file is identified as fid = <vol\_id, vnode #, unique identifier>
  - All AFS servers keep a copy of “volume location database”, which is a table of vol\_id → server\_ip mappings
  - Can move volumes between servers to balance load

# Implications on Location Transparency



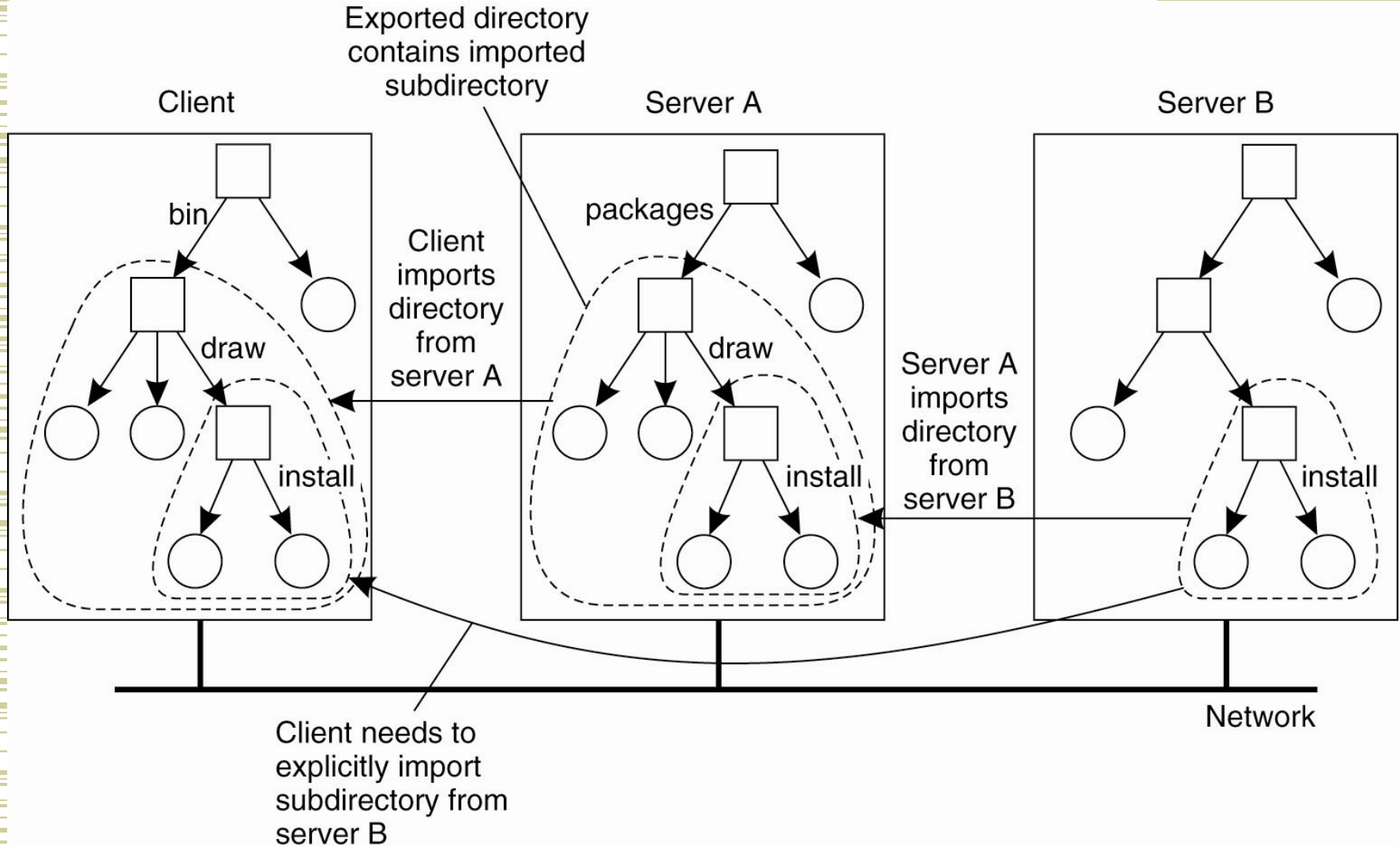
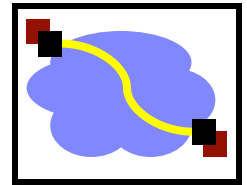
- NFS: no transparency
  - If a directory is moved from one server to another, client must remount
- AFS: transparency
  - If a volume is moved from one server to another, only the volume location database on the servers needs to be updated

# Naming in NFS (1)

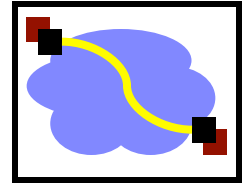




# Naming in NFS (2)

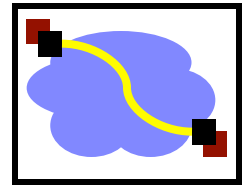


# Outline



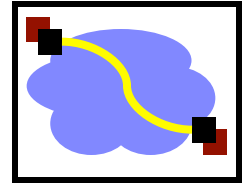
- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

# Topic 4: User Authentication and Access Control



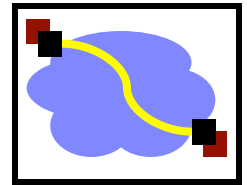
- User X logs onto workstation A, wants to access files on server B
  - How does A tell B who X is?
  - Should B believe A?
- Choices made in NFS V2
  - All servers and all client workstations share the same <uid, gid> name space → B send X's <uid, gid> to A
    - Problem: root access on any client workstation can lead to creation of users of arbitrary <uid, gid>
  - Server believes client workstation unconditionally
    - Problem: if any client workstation is broken into, the protection of data on the server is lost;
    - <uid, gid> sent in clear-text over wire → request packets can be faked easily

# User Authentication (cont'd)

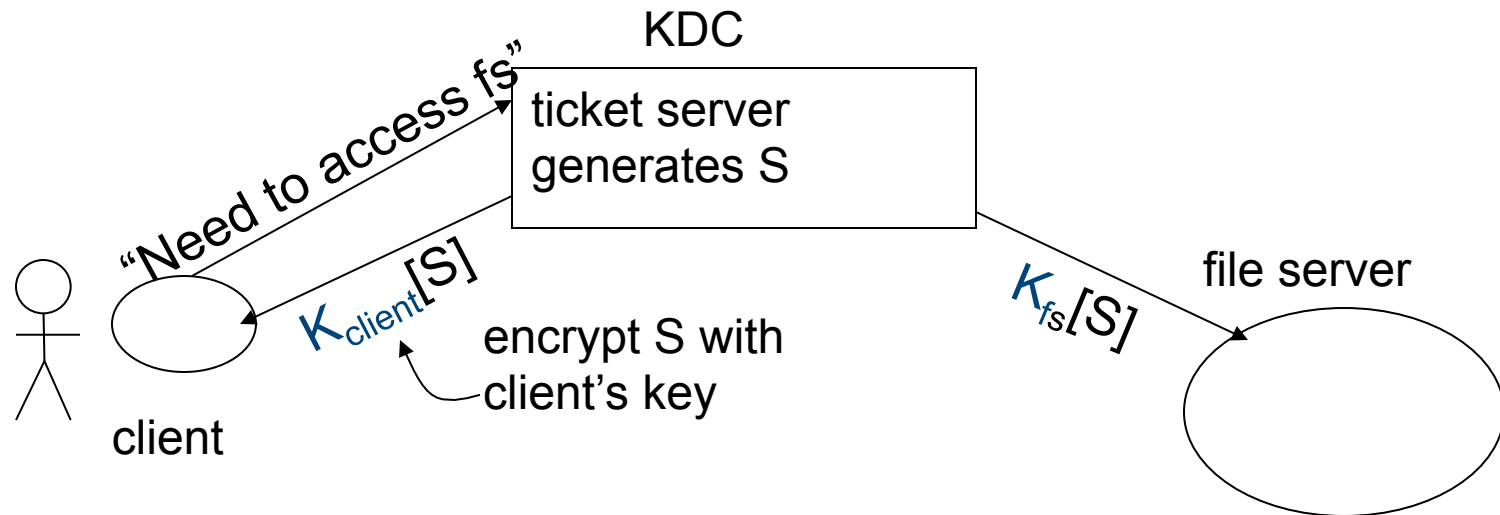


- How do we fix the problems in NFS v2
  - Hack 1: root remapping → strange behavior
  - Hack 2: UID remapping → no user mobility
  - Real Solution: use a centralized Authentication/Authorization/Access-control (AAA) system

# A Better AAA System: Kerberos

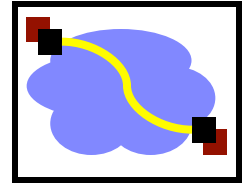


- Basic idea: shared secrets
  - User proves to KDC who he is; KDC generates shared secret between client and file server



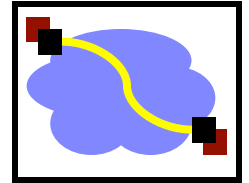
$S$ : specific to {client,fs} pair;  
"short-term session-key"; expiration time (e.g. 8 hours)

# Today's bits



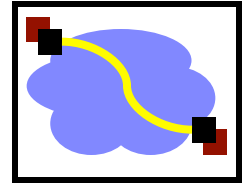
- Distributed filesystems almost always involve a tradeoff: consistency, performance, scalability.
- We've learned a lot since NFS and AFS (and can implement faster, etc.), but the general lesson holds. Especially in the wide-area.
- We'll see a related tradeoff, also involving consistency, in a while: the CAP tradeoff. Consistency, Availability, Partition-resilience.

# More bits



- Client-side caching is a fundamental technique to improve scalability and performance
  - But raises important questions of cache consistency
- Timeouts and callbacks are common methods for providing (some forms of) consistency.
- AFS picked close-to-open (session) consistency as a good balance of usability (the model seems intuitive to users), performance, etc.
  - AFS authors argued that apps with highly concurrent, shared access, like databases, needed a different model

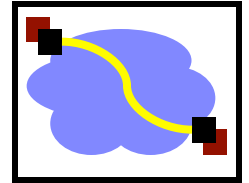
# Failure Recovery in AFS & NFS



- What if the file server fails?
- What if the client fails?
- What if both the server and the client fail?
- Network partition
  - How to detect it? How to recover from it?
  - Is there anyway to ensure absolute consistency in the presence of network partition?
    - Reads
    - Writes
- What if all three fail: network partition, server, client?



# Key to Simple Failure Recovery



- Try not to keep any state on the server
- If you must keep some state on the server
  - Understand why and what state the server is keeping
  - Understand the worst case scenario of no state on the server and see if there are still ways to meet the correctness goals
  - Revert to this worst case in each combination of failure cases