

AFS

04.23.2025

Distributed File Systems

Recap distributed system problems:

- Things can fail / timeout
- Communication methods
- Consistency
- RPC

File system concerns

- **Why** do we even want one?
 - Reliability
 - Sharing
- **How** should we interact with the filesystem? Aka, what is the **interface**?
 - User library?
 - VFS?
- What is the **consistency** model?
 - Concurrent updates w/ multiple clients
 - Server failure?
 - Client failure?
- **Where** to store distributed state?
 - Server must maintain state for each client?

NFS

Lecture Tomorrow

AFS : Andrew File System

- Objective: Scalability! (1000's of machines)
- More reasonable semantics for concurrent file access

AFS Design

- NFS: Server exports local FS
- AFS: Directory tree stored across many server machines (helps scalability!)



Break directory tree into “volumes”
I.e., partial sub trees

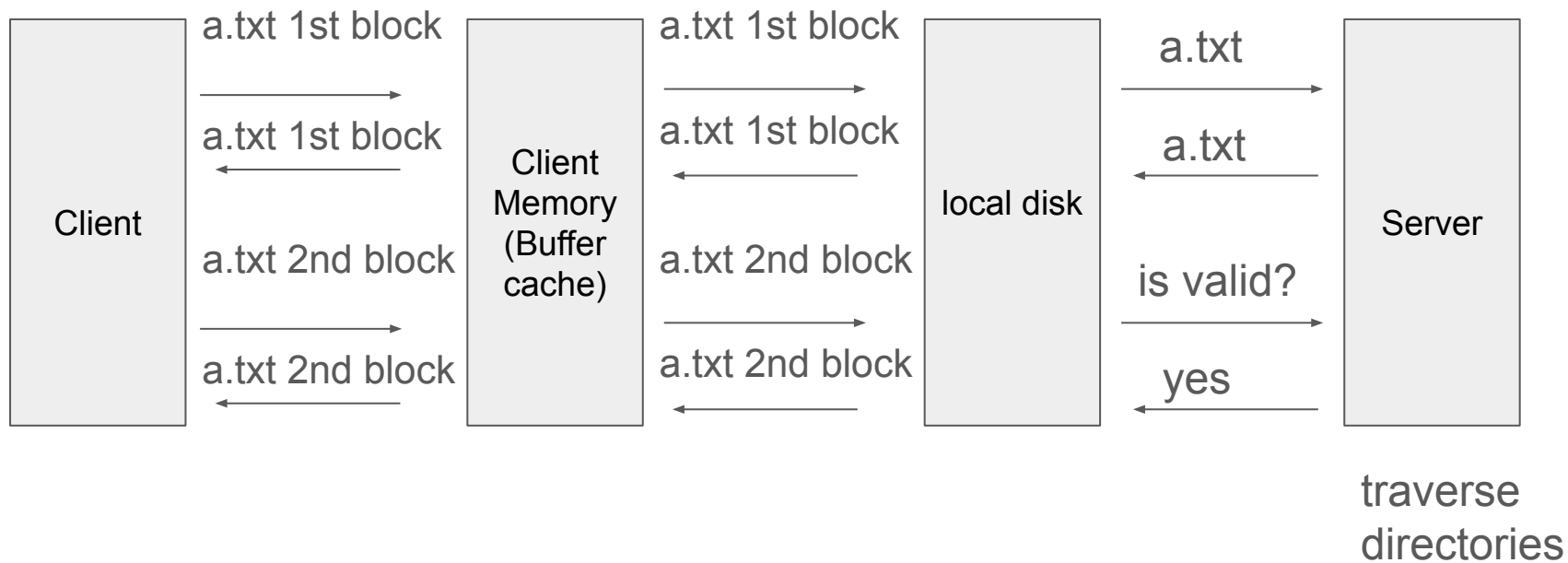
Prototype

Key idea: "whole file caching"

- Contact server during open and close
- Reads and writes performed locally (this is contrast to NFS).
- "Clients cache entire files from a collection of dedicated autonomous servers."

Requirements: we assume client has local-disk cache.

Prototype



Observations

- Works okay
- Slower than local file access, but faster than logging into timesharing system
- Some applications run slowly
 - Repetitive stat calls to check if file exists
- Dedicated process per client is very resource-intensive (Path traversal)
 - Resource limits on server
 - Context switches - Single process per client
 - Only scaled to ~20 users/server

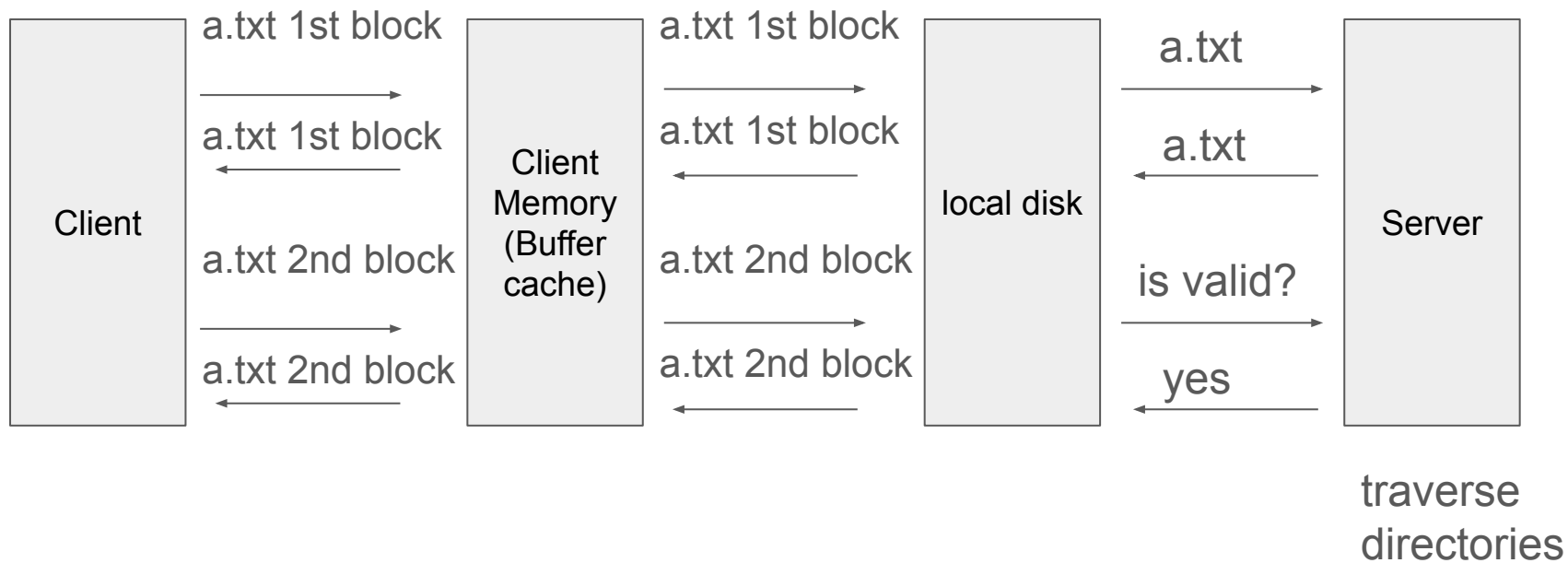
Benchmark and Breakdown

- Two RPCs made up over 90% of all RPCs
 - TestAuth
 - GetFileStat
- Look at CPU usage of each server
 - high CPU, low I/O usage.
 - more profiling: context switch cost, path traversal

Revised Version

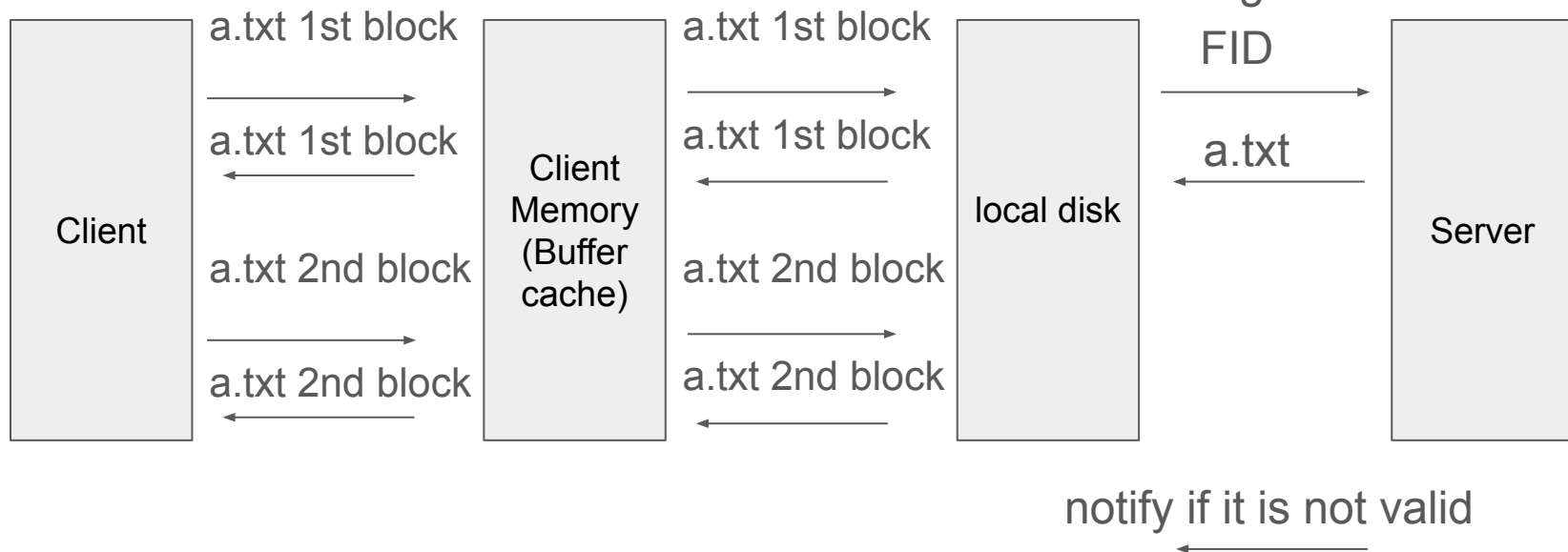
- Cache management
 - Assume cache entries are valid unless otherwise **notified**.
 - Server will make a “**callback**” function if another client changes the file.
 - What changes? Server now needs to maintain state for each client.
- Name resolution - Introduce “**file id**”.
 - Directory entries map path segments to fid.
 - Server isn't aware of path at all.
 - volume | vnode | uniquifier. (to enable reuse of the volume and file IDs when a file is deleted)
All fields are **unique** and contain info about which volume the file is located.
- Communication and server processes
 - Use lightweight processes (i.e., **threads**) instead of processes on server.
- Low-level storage representation
 - Access file by vnode (which corresponds to local inode) **instead of path**.
 - Needed to modify the local file system to do this.
 - Goal: eliminate path lookups

Prototype



Revised

- FID of 'a.txt' can be found by recursively fetching directories



Consistency

- What are the implications of open-to-close semantics?
 - Writes to a file are immediately visible to other processes on the same machine
 - Writes to a file are only visible to other machines after the file is closed
 - Last writer wins.
i.e. no filesystem locking.
Clients have to coordinate themselves if they want locks.

Consistency

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 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		A	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		C	D	
		D	open(F)	D	D	Unfortunately for P ₃ the last writer wins
		D	read() → D	D	D	
		D	close()	D	D	

Figure 50.3: Cache Consistency Timeline

Evaluation

