THE UNIVERSITY *of* EDINBURGH

Distributed Systems Fall 2024

Yuvraj Patel

Today's Agenda

Distributed File systems

- Network File System (NFS)
- Andrew File System (AFS)

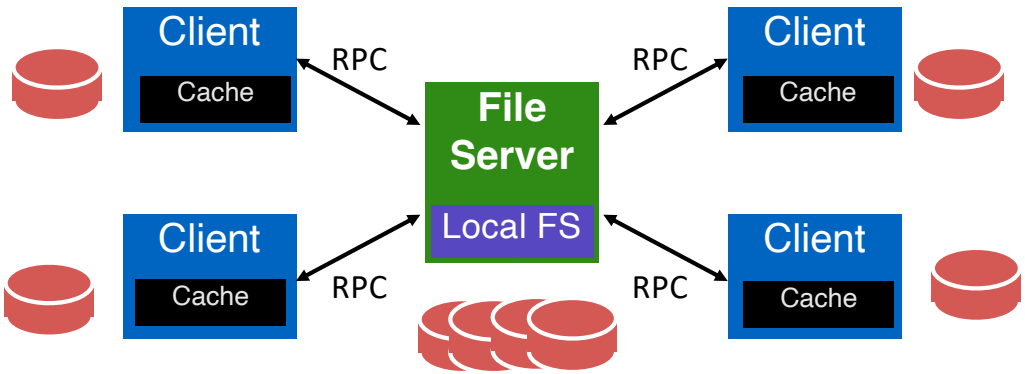
Google File System (GFS)

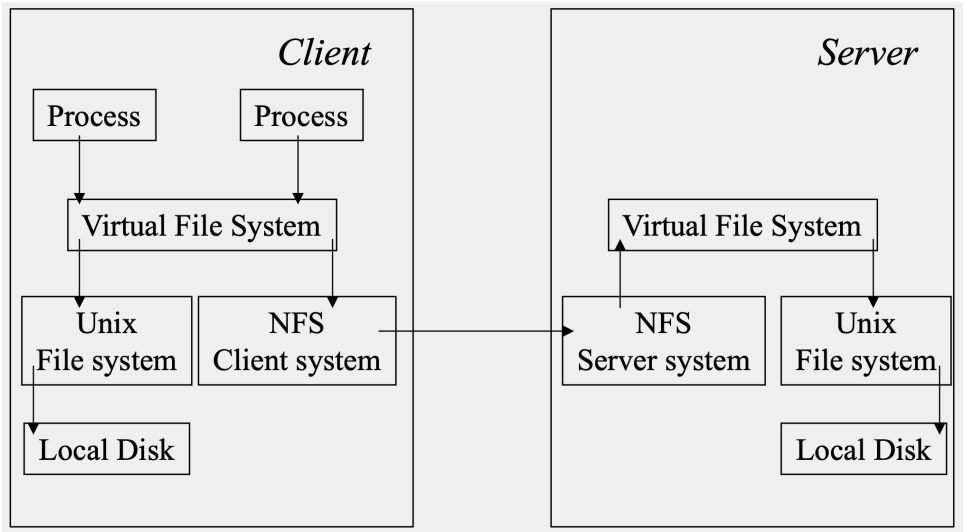
Next (Last) Class Tuesday (26/11)

Course Feedback -- https://forms.office.com/e/mCiERR672z

Demo slots finalized – Check piazza

NFS Architecture





RPC: Remote Procedure Call
Cache individual blocks of NFS files

What do Clients Send to Server?

Stateless Protocol + File Handle + Client Logic

Build normal UNIX API on client side on top of RPC-based APIs

Clients maintain their own file descriptors

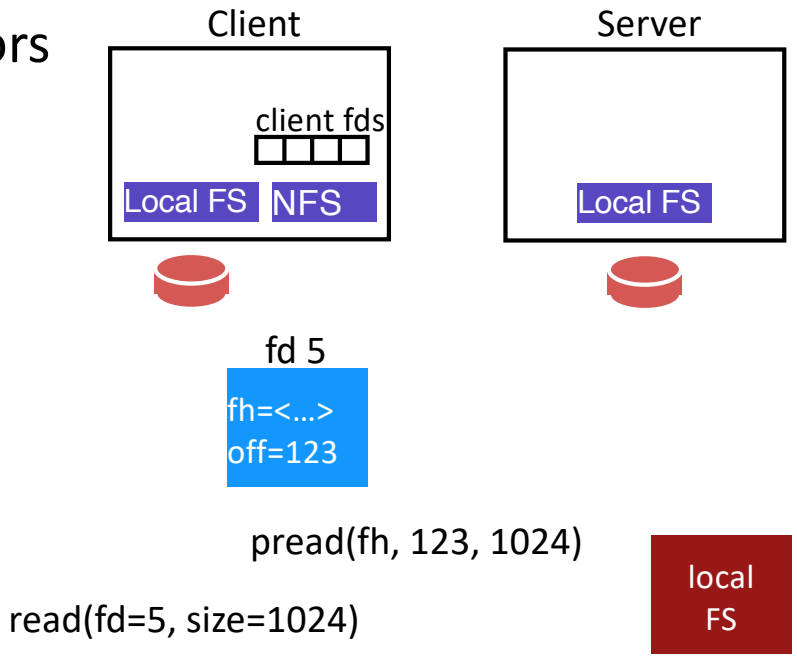
Client open() creates a local fd object

Local fd object contains

- File handle (returned by server)
- Current offset (maintained by client)

Client sends fh, offset, size to server

Server extracts inode from fh



Idempotent vs. Non-Idempotent Operations

Idempotent Operations

- If f() is idempotent, f() has the same effect as f(); f(); …. f(); f();
- pwrite(), any read operation

Non-Idempotent Operations

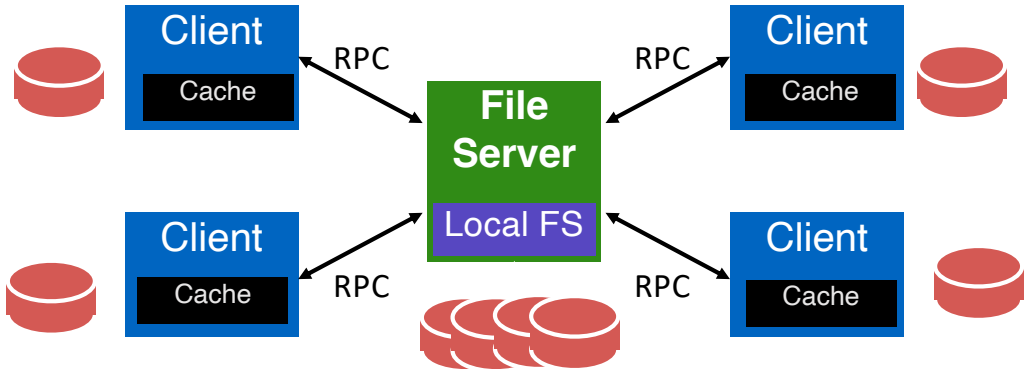
- Cannot be retried multiple times
- Append, mkdir, rmdir, creat

NFS Caching

With NFS, data can be cached in three places

- Server memory
- Client disk
- Client memory

How to make sure server and all client versions are in sync?

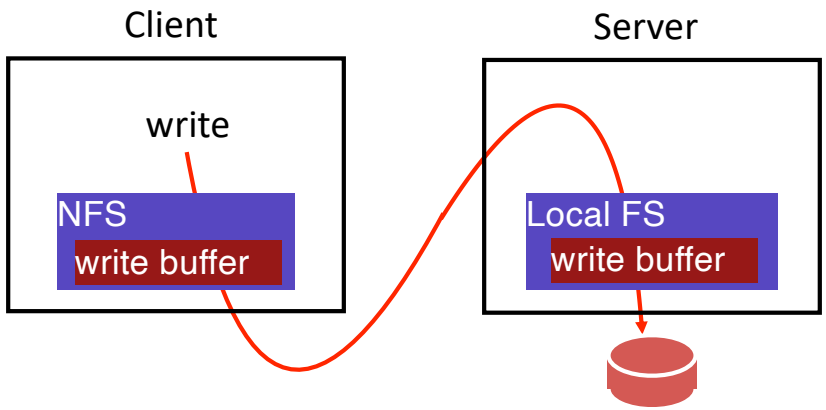


NFS Caching: Problem 1

NFS server often buffers writes to improve performance

Server might acknowledge write before pushed to disk

What happens if server crashes?



NFS Caching: Problem 1 (contd…)

NFS server often buffers writes to improve performance

Server might acknowledge write before pushed to disk

What happens if server crashes?

Solutions:

- Don’t use server write buffer (persist data to disk before acknowledging write) → Slow
- Use persistent memory → More expensive

NFS Caching: Problem 2

Clients must cache some data

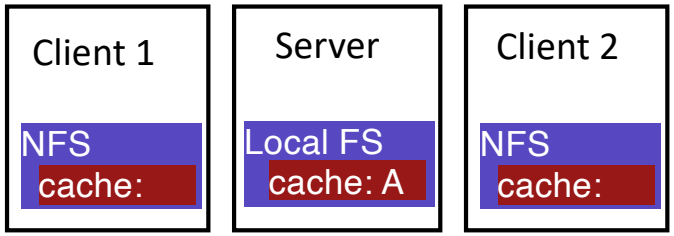
- Too slow to always contact server;
- Server would become severe bottleneck

Update visibility problem: Server doesn't have latest version

Some clients may see old version (different semantics than local file system)

When client buffers a write, how can server see update?

- Client flushes cache entry to server
- When should client perform flush?



NFS Caching: Problem 2 (contd…)

When should client perform flush?

Possibilities

- After every write (too slow)
- Periodically after some interval (odd semantics)

NFS Solution

- Flush on close()
- Other times optionally too – e.g., when low on memory

Problems not solved by NFS

- File flushes not atomic (one block of file at a time)
- Two clients flush at once can lead to mixed data

NFS Caching: Problem 3

"Stale Cache" Problem

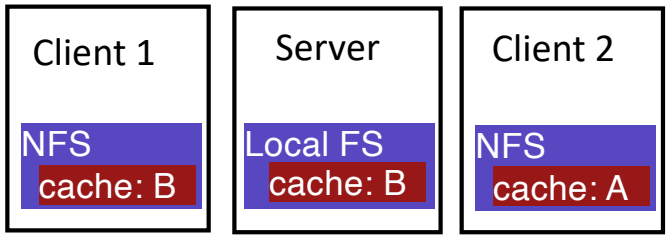
- Clients doesn't have latest version from server
- Clients may see old version (different semantics than local FS)

How can it get latest update?

- Maintaining state – push update to relevant clients

Stateless solution

- Clients recheck if cached copy is current before using data
- Recheck faster than getting data



NFS Caching: Problem 3 (contd…)

Client cache records time when data block is fetched (t1)

Before using data block, clients sends file STAT request to server

- STAT gets last modified timestamp (t2) for this file

If t2 > t1, then refetch data block

NFS developers found server overloaded

- Found stat accounted for 90% of server requests

Fix

- Client caches result of stat (attribute cache)
- Make stat cache entries expire after a given time (3 seconds)
- Clients could read data that is up to 3 seconds old

Andrew File System (AFS)

Andrew File System: Developed at CMU in 1980s

Used in many universities (UoE home directories are AFS backed)

Goals

- More reasonable semantics for concurrent file access
- Improved scalability (many clients per server)
- Willing to sacrifice and statelessness

AFS Whole File Caching

Approach

- Measurements show most files are read in entirety
- open(): AFS client fetches whole file, storing in local memory or disk
- close(): Client flushes file to server if file was written

Convenient and intuitive semantics

- Use same version of file entire time between open() and close()

Performance advantages

- AFS needs to do work only for open/close (less load on server)
- Reads/writes are completely local

AFS Caching

AFS faces same problem as we discussed with NFS

Update Visibility

- How are updates sent to the server

Stale Cache

- How are other caches kept in sync with server?

AFS Caching – Update Visibility

AFS, like NFS, also flush on close

Buffer whole files on local disk; update file on server atomically

But what about concurrent writes?

- Last writer wins (i.e., the last file close wins)
- Newer get data mixed from multiple versions on server unlike NFS

AFS Caching: Stale Cache

Stateful solution unlike NFS' stateless solution

Server tells clients when data is overwritten

- Server must remember which clients have the file open right now

When clients cache data on open(), ask for "callback" from server if file changes

- Clients can use data during this open() without caching

Clients only verifies callback when open() file (not every read)

- May not refetch file on next open()
- Operate on same version of file from open to close

AFS Callbacks: Dealing with State

Callbacks are good to handle the stale cache issue.

What about client and server crashes?

AFS Callbacks: Dealing with State (contd…)

Client crash

- After reboot, cached data might be on client disk
- Might read stale data from the cached copy
- Solutions
  - Evict everything from cache
  - Recheck specific entries before using

Server crash

- Lose track of all clients who have file open
- Solution – Tell all clients to recheck all data before next open

NFS vs AFS Protocols

Time	Client A	Client B	Server Action?
0	fd = open("file A");		
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	
50		write(fd, block1);	
60	read(fd, block1);		
70		close(fd);	
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

NFS Protocol

Time	Client A	Client B	Server Action?
0	fd = open("file A"); Filehandle		Lookup for file A
10	read(fd, block1);		Read
20	read(fd, block2);		Read
30	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy		Get_attr()
31	read(fd, block2); Latest attributes		Get_attr()
40		fd = open("file A"); Filehandle	Lookup for file A
50		write(fd, block1); Keep local	
60	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
70		close(fd); Send data to server	Write to disk
80	read(fd, block1); Check cache; attr expired; call get_attr(); expired; flush cache; fresh read again		Get_attr()
81	read(fd, block2); Latest attributes		Get_attr()
90	close(fd);		
100	fd = open("fileA"); Filehandle		Lookup for file A
110	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
120	close(fd);		

AFS Protocol

Time	Client A	Client B	Server Action?
0	fd = open("file A");		Setup callback for A, send all of file A
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1); Local read		
31	read(fd, block2);		
40		fd = open("file A");	Setup callback for A, send all of file A
50		write(fd, block1);	
60	read(fd, block1); Local read		
70		close(fd); Send back changes of A; Server break call backs	
80	read(fd, block1); Local read		
81	read(fd, block2); Local read		
90	close(fd);		
100	fd = open("fileA"); No callback; fetch file A again		Setup callback for A, send all of file A
110	read(fd, block1); Local read		
120	close(fd);		

When will server be contacted for NFS? For AFS?
What data will be sent? What will each client see?

Google File System (GFS)

Scalable, distributed file system for large distributed data-intensive applications at Google

Based on a different set of assumptions leading to different design choices than conventional file systems

Implemented as user-space library

Goals

- Large storage
- Scalable access
- Fault tolerance
- Transparency (location and fault)

GFS Assumptions

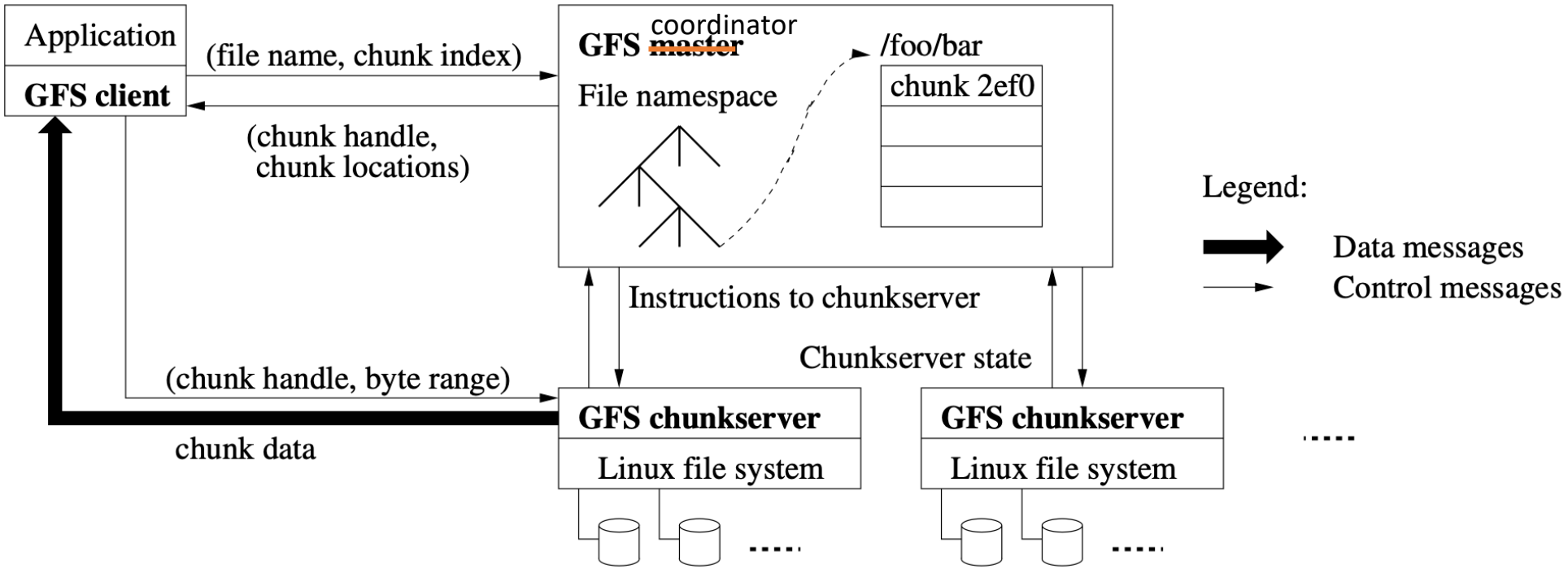
Hardware failures are common

Files are large (GB/TB); millions of files

Files access

- Most files are appended, not overwritten; Large appends
- Random writes within a file are almost never done
- Files are mostly read; often sequentially
- Two types of reads – Large streaming reads and small random reads
- Concurrent appends possible

GFS Architecture



GFS Architecture – Coordinator

Single Coordinator

- State replicated on backups

Holds all metadata

- Namespace
- Access-control
- Filename to Chunk Mapping
- Chunk locations

All metadata stored in the main memory

GFS Architecture – Coordinator (contd…)

Manages

- Chunk leases to chunkservers
- Garbage Collection of orphaned chunks
- Chunk migration (copying/moving chunks)

Fault Tolerance

- Periodically communicates with all chunkservers via heartbeats
- Operation log replicated on multiple machines

GFS Architecture – Chunk & Chunkservers

Chunk size = 64 MB

Chunkserver

- Stores chunks on local disks as normal files
- Stores a 32-bit checksum with each chunk to detect corruption

Each chunk replicated (3 replicas) on multiple chunkservers

Chunk Handle to identify a chunk

- Globally unique 64-bit number
- Assigned by the coordinator when the chunk is created
- Read/write requests specify chunk handle and byte range

GFS Files Viewpoint

GFS Clients

Hundreds/Thousands of clients

Issues

- Control requests to coordinator
- Data requests directly to chunkservers

Caches metadata

No caching of data

No OS-level API; instead use library (GFS client code linked into each application)

GFS Client Read

GFS Client Write

One chunkserver is primary for each chunk

Coordinator grants lease to primary (60 sec)

Leases renewed using periodic heartbeat messages between coordinator and chunkservers

Primary chooses the order for all client writes

- Tells the secondaries – with sequence numbers – so all replicas apply writes in the same order, even for concurrent client writes

GFS Client Atomic Record Append

GFS provides an atomic append operation called record append

Unlike traditional writes, client specifies only the data

GFS appends it to the file at least once atomically at an offset of GFS's choosing and returns that offset to the client

- Like writing to a file opened in O_APPEND mode in Unix without race conditions when multiple writers do so concurrently

GFS Client Atomic Record Append (contd…)

Same control flow as writes

Client pushes data to replicas of last chunk of file

Client sends request to primary

Request fits in current last chunk

- Primary appends data to own replica
- Primary tells secondaries to do same at same byte offset in theirs
- Primary replies with success to clients

GFS Client Atomic Record Append (contd…)

If data does not fit in the last chunk

- Primary fills current chunk with padding
- Primary instructs secondaries to do the same
- Primary replies client to retry the operation on next chunk

If record append fails at any replica, client retries operation

- Replicas of same chunk may contain different data including duplicates of all or part of record data

GFS Metadata Consistency & Operation Log

Changes to namespace are atomic

Coordinator uses operation log to store critical metadata changes

Log defines a timeline that defines the order of concurrent operations

Log stored on coordinator's local disk and replicated on remote machines

Coordinator recovers its file system state by replaying the operation log

Coordinator only replies to client after log entries safe on local disk and replicas

GFS Consistency Model – Data

Defined – If primary tells client that a write succeeded, and no other client is writing the same part of the file, all readers will see the write

Consistent – If successful concurrent writes to the same part of a file happens, and they all succeed, all readers will see the same content, but maybe it will be a mix of the writes

Inconsistent – If primary doesn't tell the client that the write succeed, different readers may see different content, or none

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

GFS Consistency Model – Data (contd…)

How can inconsistent content arise?

- Primary updates its own state but one secondary did not update (slow, fail)
- Client1 reads from P; Client 2 reads from S1
- Both clients will see different results

How can consistent but undefined arise?

- Clients break big writes into multiple small writes (at chunk boundaries), and GFS may interleave them if concurrent client writes happen

How can duplicate data arise?

- Clients re-try record appends

Applications must cope with the data inconsistency

GFS Applications & Record Append Semantics

Applications rely on checksum in records they write using Record Append

Readers can identify padding and record fragments using checksum

If application cannot tolerate duplicates, they can use unique identifiers in the records

Readers can use unique identifies to identify and filter duplicates

GFS Stale Replica Detection

For each chunk, coordinator maintains a chunk version number to distinguish between up-to-date and state replicas

Coordinator increase chunk version number and informs the up-to-date replicas when the coordinator grants a new lease on a chunk

When chunkserver restarts/recovers after crash, on reporting its set of chunks and their version numbers, coordinator can identify stale replicas

Coordinator removes state replicas in its regular garbage collection

Coordinator also passes the version number to client; clients can check the version numbers to ensure it is accessing up-to-date data

GFS – Handling faults

Secondary

- Primary may retry “n” times
- Client can retry
- Coordinator may remove from chunkhandle lists, replicate chunk data

Primary

- Coordinator may remove from chunkhandle lists
- Coordinator will grant lease to any secondary

Coordinator

- Replay operation log, rebuild state, resume operations
- Ask chunkservers what they store
- Wait for one lease time before granting lease to any secondary

End of Lectures