CS 423
Operating System Design: Remote Access & GFS
Apr 30

Ram Kesavan

LOGISTICS

Midterm papers: pickup from my office during OH

MP4 due by May 6, 11:59 CT

Last lecture: 5/5 will be a review

Will publish some practice questions next week

Finals: 5/13 at 7-10pm, DCL 1320 (next door)

AGENDA / LEARNING OUTCOMES

Remote Access of File Systems

- Wrap-up NFS
- Mention other protocols

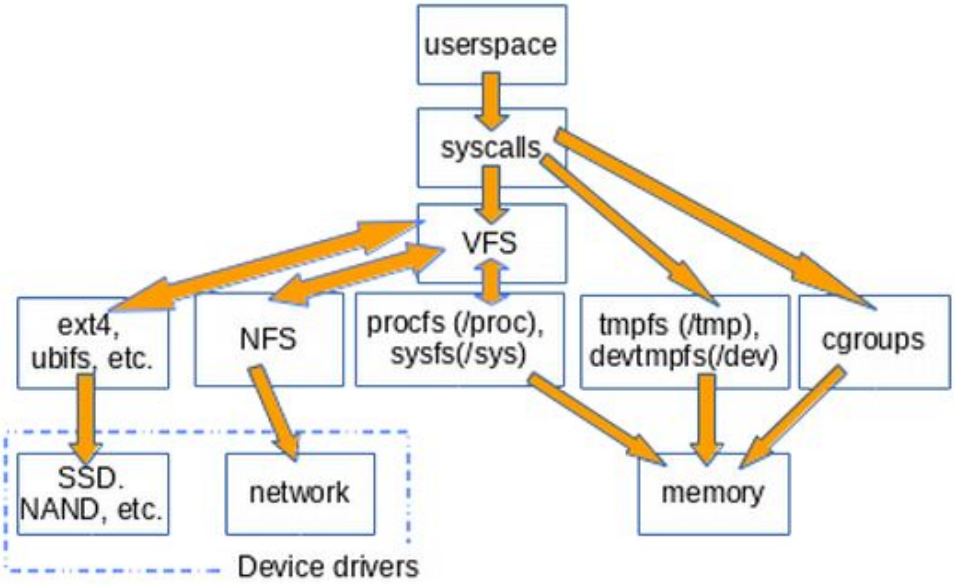
Data Integrity

Distributed File Systems

- GFS

RECAP

VIRTUAL FILE SYSTEM LAYER



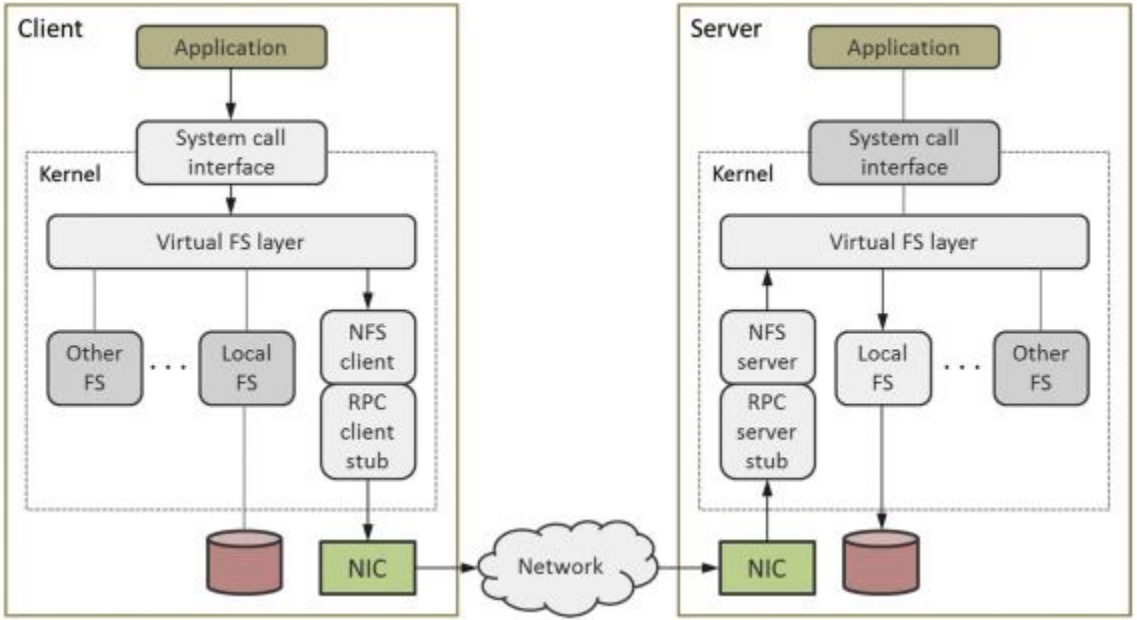
Local

- File systems on locally attached HDDs & SSDs
- File systems on local DRAM: do not survive crashes
- pseudo file systems

Remote

- Protocol-specific client that talks with remote file service
- Protocols:
  - NFS: Sun
  - SMB: Microsoft
  - AFS: CMU (legacy)

NETWORK FILE SYSTEM (NFS)



OSTEP Chapter 48

- Client-server model
- Client allocates a socket and establishes communication over TCP or UDP
- Uses remote procedure call (RPC) abstraction to send & receive messages
- NFS protocol messages

OPEN SYSCALL: STATELESS

- Stateless NFS: A file identifier that survives server restarts

Client A

1. fdA = open("foo/bar", O_READONLY);

9. read(fdA, buf, 4096);
// NFS read with <v, 501, g>

Client B

3. unlink("foo/bar");

5. fdB = open("foo/baz", O_CREAT);

7. write(fdB, buf, 4096);

Server

2. // inum of "bar" = 501
respond with <v, 501, g>

4. // inum 501 is now free

6. // 501 is reused for baz
respond with <v, 501, g+1>

8. ack

10. responds with ESTALE

END RECAP

NFS V2 CLIENT⟶SERVER MESSAGES

`open()`: LOOKUP(s) to walk path + GETATTR
    O_TRUNC: SETATTR (to truncate file to zero length)
    O_CREAT: CREATE (to create a new file)
`read()`: READ
`write()`: WRITE
`unlink()`: REMOVE
`readdir()`: READDIR
    returns (subset of) dir entries (+ opaque resumption cookie)
`mkdir()`: MKDIR
`rmdir()`: RMDIR
`close()`: nothing!*

*client-side caching

NFS V2 IDEMPOTENCY

Several failure possibilities

- NFS request lost
- Server goes down/restarts
- NFS response is lost

Idempotency simplifies failure handling

- Client retries request after timeout
- Non-modify ops are trivially idempotent
- WRITE is also idempotent
- But, life is not perfect
  - REMOVE, MKDIR, RMDIR

PERFORMANCE: CLIENT-SIDE CACHING

Can cache data & metadata

Obvious benefit for non-modify ops

Also cache results of write (aka **write buffering**)

`fsync()` forces a flush to server

2 Problems

- ClientA's read can't see results from ClientB's write

  flush-on-close semantics: `close()` flushes buffered writes

  only subsequent reads see updated content

- Stale content in ClientA's cache

  client sends GETATTR to server

  response includes file's last-modified-timestamp

  check if its equal to attributes in client's cache

  if not, invalidate cache content & re-fetch file from server

PERFORMANCE: SERVER-SIDE CACHING

Server caches data & metadata

- Can respond without accessing storage devices

Problem?

- Delay in persisting modify ops
- Solved by journaling

NFS has been a huge success

- Is the dominant file access protocol
- Many companies: Sun, NetApp, EMC, IBM sell/sold NFS servers

OTHER PROTOCOLS

NFS versions (v2 is now obsolete)

- v3: better performance, larger file sizes (>2GiB)
- v4: stateful; connection-oriented, improved security, locking
- v4.1: pNFS - client can read/write directly from multiple servers
  works with distributed NFS service

AFS: built by CMU, now obsolete; Ch50 of OSTEP

- Client open( ): **fetches + caches entire file on local storage!**
  - Contrast: NFS caches only fetched blocks of file in memory

Stateful protocols: SMB, AFSv2, NFSv4+

- Connection-oriented
- Server tracks clients, open files & locks
- Server uses callback to inform a client
  - eg. your cached content is stale

NEXT

Data Integrity

Distributed File Systems

GFS + Colossus

DATA CORRUPTION

Sometimes, bytes get damaged silently
- group of sectors is touched by a disk head
- cosmic rays flips some bits!

How to detect SDC (silent data corruption)?
- checksum per block
- each write: compute & store checksum with each block
- each read: compute checksum & compare with the stored one

If detectable, easier to handle: use RAID
- checksum mismatch
- entire storage device dies (fail-stop model)
- device returns error on operation

The problem of Lost/Misdirected Write!
- YT demo lecture: in the Misc Resources section of the course webpage

WHY DISTRIBUTED?

Limits of using a single server

- Total storage attached to that server
- CPU + memory for file system operations → performance
- Availability (when server goes down)
- And, finally cost!

Alternatively:

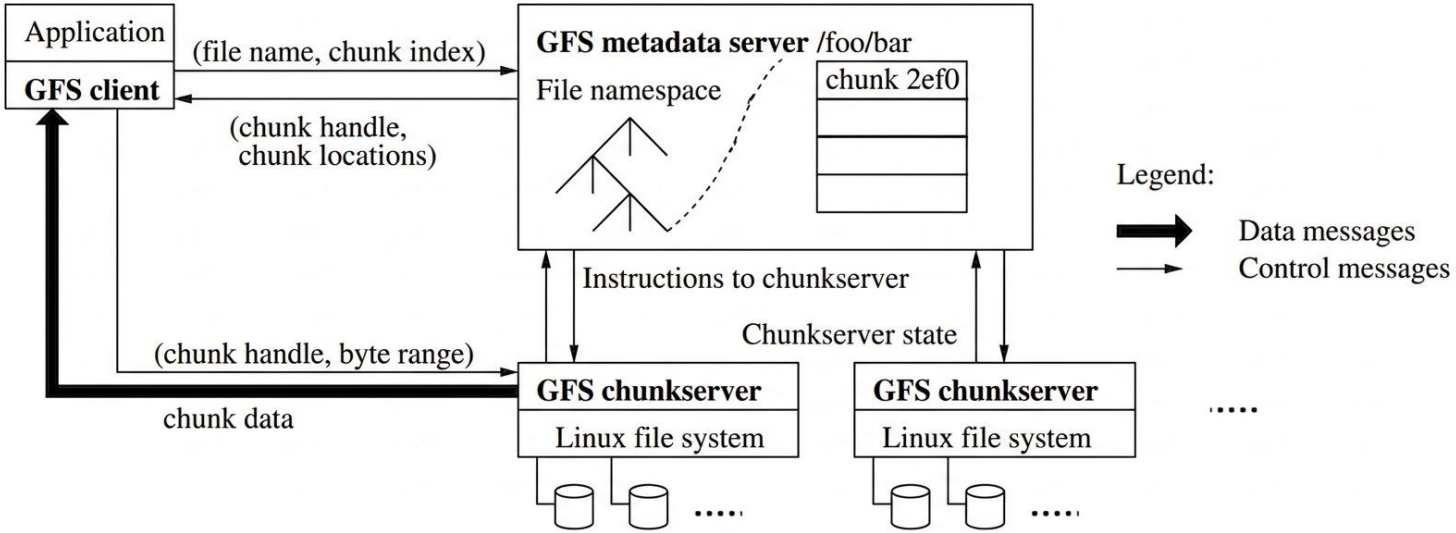
- File system not limited by server count
- aka "scale-out": can keep adding more cheap/commodity servers
- Transparency: clients unaware about location, failure, replication, etc.
- Ideally, the IOPS load & storage gets evenly distributed across servers
- Allows for pNFS style high-throughput access to a file's data
- Designed for failures
  - Expected when system comprises 100s-1000s of cheap servers

MANY POPULAR SYSTEMS

Most (all?) of these file systems run in user-space

Name	Intro Yr	Open Source?	Company	POSIX?
GPFS	1998	No	IBM	Yes
GFS	2003	No	Google	No
HDFS	2006	Yes	Yahoo	No
CephFS & GlusterFS	2006	Yes	Red Hat	Yes
Pangu	2009	No	Alibaba	No
Tectonic	2021	No	Meta	No

GFS



Application is compiled with GFS client; client hides all complexity
Single metadata server + many data chunkservers
Chunkserver uses Linux file system to store the chunks

GFS ARCHITECTURE

GFS: designed for reading/writing large files

No inodes in GFS!

- Flat map of name+offset→chunkID
- File: is just a list of chunks, each chunk gets a 64-bit ID

Single metadata server; can be a bottleneck

- Solved by Colussus (GFS successor); uses many metadata servers

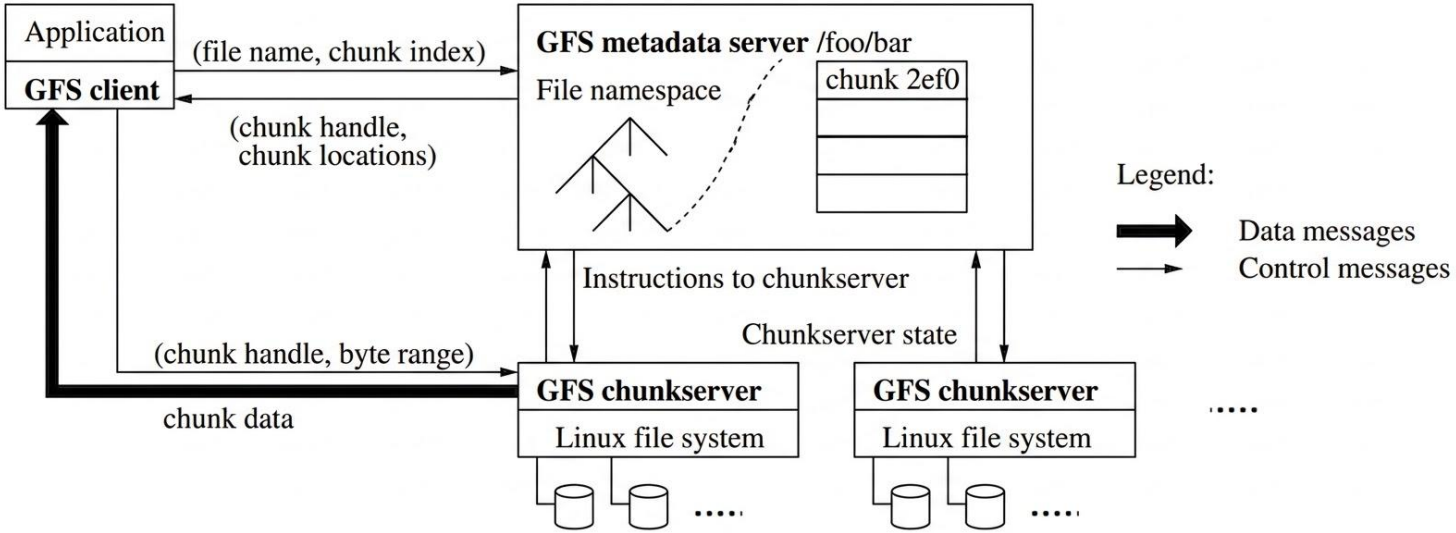
Chunk server: stores chunk in its local FS like ext4

- 3 replicas of each chunk; each stored in a different chunk server

Heartbeats between metadata & chunk servers

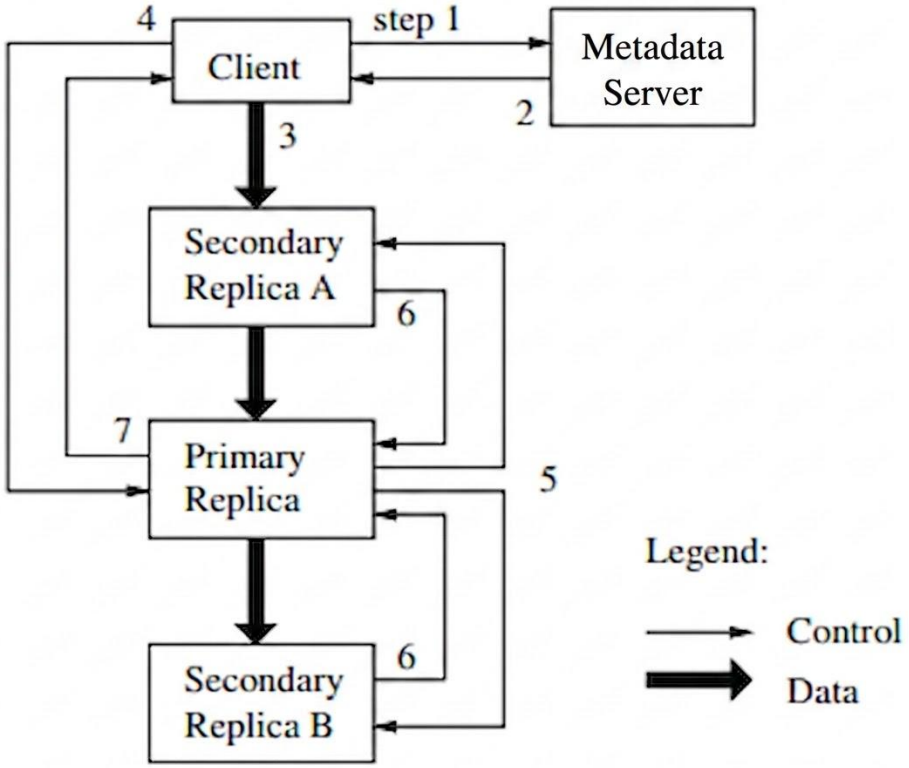
Chunk size: 64 MB

GFS: READ PATH



Client→MS: server filename + offset
MS→client: chunk-handle + replica-locations
Client caches metadata information
    Client picks a chunkserver (typically, closest) and directly reads from it
Future reads: client uses cached metadata, directly communicates with chunk server

GFS: WRITE PATH



1. Client→MS: replicas for chunk
2. MS→client: primary+2ndary
   Client caches metadata info
3. Client→all replicas: data in parallel
   Waits for all replicas to ack
4. Client→primary: this write
   Primary gives it a serial#
   (across all writes to that chunk)
5. Primary→replicas: this serial#
   replicas apply writes in serial order
6. Replicas→primary: ack
7. Primary acks client

If any failure, client is informed
Client lib retries the write op

INTERESTING DETAILS

Files **are append-only**

- Files get written once sequentially
  - COW: When updated, a new version gets written out
  - Supports file versioning
- Simplifies GFS design

Erasure encoding across chunk servers

- 3 (or more) replicas gets expensive: 3x storage
- Reed-Solomon encoding (<1.5x storage)

Not discussed

. file snapshots
. file deletion
. garbage collection
. replica management
. load/storage rebalancing
. performance:
  caching
  SSDs & HDDs
. data integrity
. failures & recovery

AFTER GFS

HDFS has similar limitations

- Single metadata server

Colossus, Tectonic, etc.

- Metadata service is also distributed & scales out

Reed-Solomon erasure encoding for data resiliency

- Uses Galois Field math
- RS(n,k): k data shards + (n-k) parity shards
- Tolerates and recovers up to (n-k) failed data shards

POPQUIZ 6

https://forms.gle/ZQNruSVCTvbV6yuXA