CS 423
Operating System Design: Persistence: AFS/GFS
05/07

Ram Alagappan

Recap

Distributed file systems, focus on two popular ones:

- NFS (network FS)
- AFS (Andrew FS)

Network file system (NFS)

- Stateless crash recovery protocol
- Cache consistency
- Open-market protocol

AFS: whole file caching vs. block caching in NFS

Logistics

Final exam:05/09 Friday

Take home

Same format at Midterm

Four questions

- material from MP and lectures
- only material after Midterm

AFSv1

Whole-file caching instead of block caching

Keep file on disk (not memory like NFS)

Open a file "/path/to/file"– fetch it from the server

Writes and reads – no interaction with server… purely local

Close a file – send it to the server

Next open – use TestAuth message to check if file is up-to-date, if yes use cached copy else fetch again

AFSv1 Problems

Path traversal costs – too much CPU to convert paths to inodes

Too many TestAuth message – most unnecessary…why?

Load imbalance across servers

Server process/threads structure

AFSv2 - Callbacks

How to reduce TestAuth messages

Callback – promise from server to client that it will inform when the file changes

Implications – now stateful!

AFSv2

fd = open(/foo/bar.txt, …)

fetch(/ FID, "foo")

Look for "foo" in /

create callback for foo for client

return foo's content and FID

write foo to disk

record callback of foo

fetch(foo FID, "bar.txt")

Look for "bar.txt" in foo

create callback for bar.txt for client

return bar's content and FID

write bar.txt to disk

record callback of bar.txt

local open cached bar.txt

return fd to app

AFSv2

read(fd…) – local

write(fd…) – local

close(fd) – flush to server using Store message – send whole file

Next open…

fd = open("/foo/bar.txt", ...);

if (callback(foo) == VALID)

    use local copy foo

else Fetch

if (callback(bar.txt) == VALID)

    open local cached copy

    return file descriptor

else Fetch then open and return fd

End Recap

Cache consistency

Update visibility: when?

Processes on different machines

- upon close, flush to server
- visible to other clients now

Processes on the same machine

- immediately visible

Last closer wins – one consistent version uploaded – compare to NFS?

Cache consistency

Invalidate stale cache: when?

after server gets latest, breaks callback

Is it possible for a client to use an older copy after the file has been changed at the server?

Is it possible for a client to open an older copy?

Crash Recovery

Client crashes – how to handle?

Treat cache as suspect – should throw cached files?

Server crashes

Keeps callback state in memory – can lose, how to handle?

use heartbeats … if not heard from server, suspect cache

Performance Comparison

N = blocks, Lnet, Lmem, Ldisk – latency of net, mem, and disk

	NFS	AFS
Small file, sequential read		
Small file, sequential re-read		
Large file, sequential read		
Large file, sequential re-read		
Large file, single read		
Small file, sequential write (new file)		
Large file, sequential write (new file)		
Large file, sequential overwrite		
Large file, single overwrite		

GFS

A file system built at Google

very successful at Google – many apps use it (e.g., search, analytics)

Published at SOSP 2003

Why learn about this FS?

- very influential – shows how to build a large-scale distributed fs
- a successful system that worked very well
- a few ideas very different from the academic world at the time (e.g., weak consistency is OK, single master is OK)

Assumptions

Failures are the norm

Fewer huge files, not many small files

Large streaming reads, some small random reads

Sequential writes that append, data once written rarely changes

Focus on throughput, not latency

Need well-defined semantics with concurrent writes, failures

Hardware at the time

100 Mbps network

12.5 MBps

Disks: 40 MBps

Aggregate bandwidth: 40 MB * 1000 machines = 40000 MBps

~ 40GBps (how many machines/ssds can give this today?)

Interface

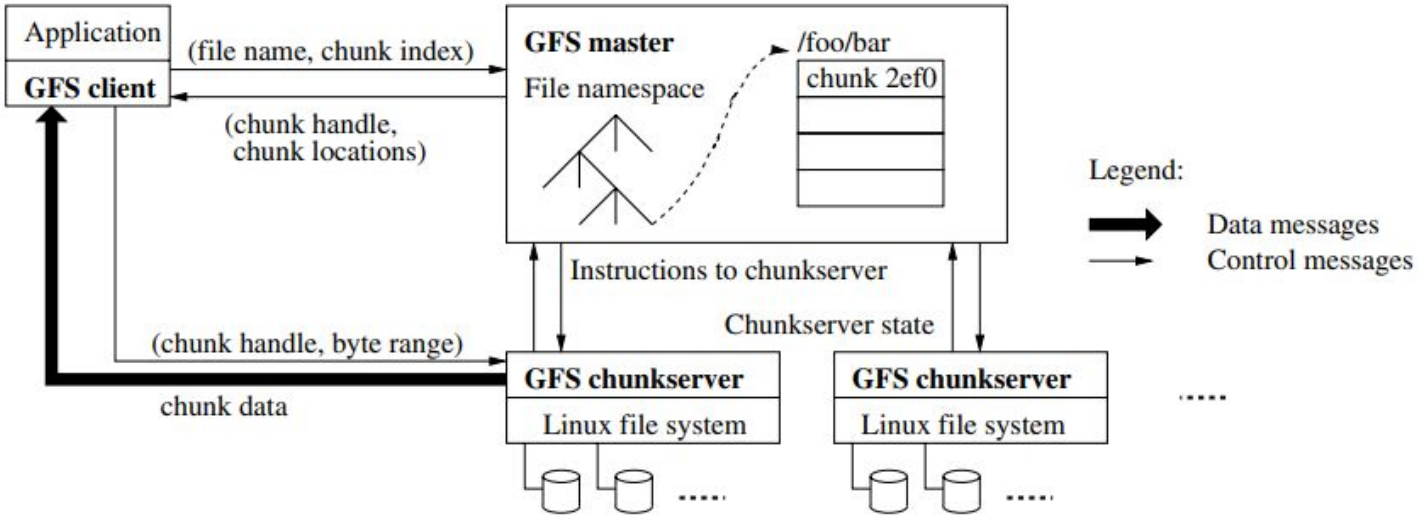
Standard FS interface

creat, delete, open, read, write, close

snapshot – create copy of file at low cost

record append – multiple clients appending to one file, ensures atomicity for each append, produce-consumer use cases

High-level Architecture



High-level Architecture

File: a bunch of chunks, chunk handle 64-bit ID

Chunk servers: data nodes, 3-way replication, use local FS like ext4

Q: cache chunks in memory?

Client: apps link a GFS-client library, hides some details from clients

e.g., client issues reads to byte ranges, client-lib and master – determine which chunk

Q: data caching at the client?

Q: what should be cached?

High-level Architecture

Master: all metadata (namespace, file □ chunk map, chunk □ locations)

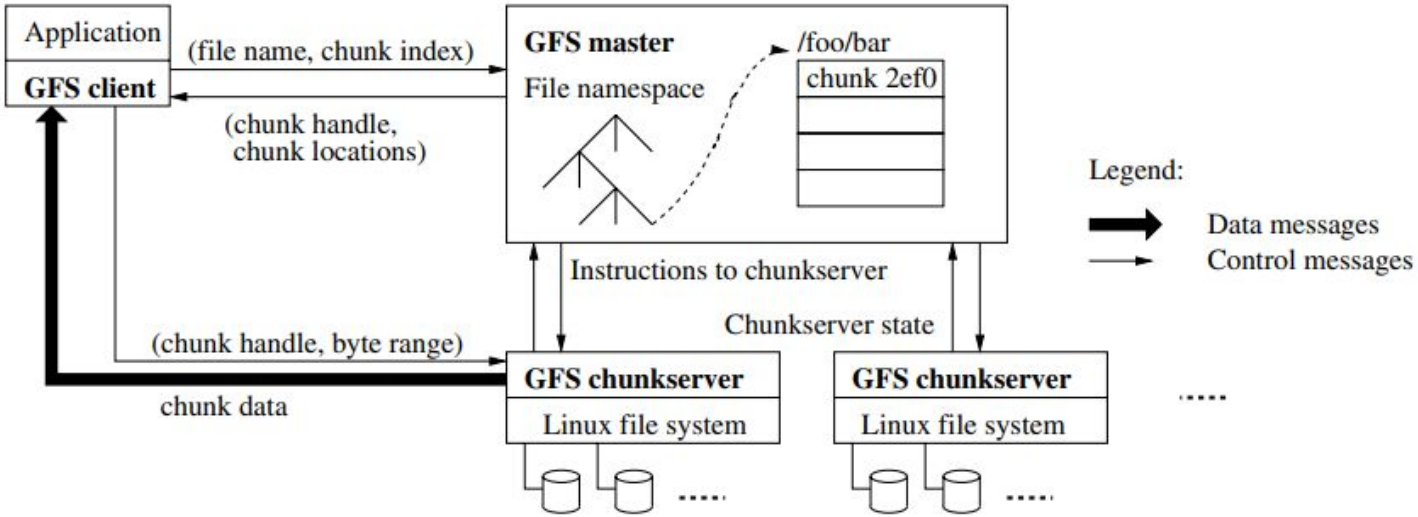
sends heartbeats to CS periodically, not involved in data ops

Unique design point: single master!

Adv: can use global knowledge, easy to manage

Q: do chunk servers know of *filenames*?

A Simple Read Operation



Chunk Size

64 MB – too large?

Benefits:

- fewer master communications
- makes chunk location maps cacheable at the clients
- less metadata on master, fit in DRAM
- better sequential bw

Problems:

- internal fragmentation?
- small file (single-chunk file) can create hotspot

Metadata

Namespace (persistent – reload from disk upon crash)

Filename □ list of chunks (persistent – reload from disk upon crash)

Chunk id □ list of servers

Q: store this map persistently?

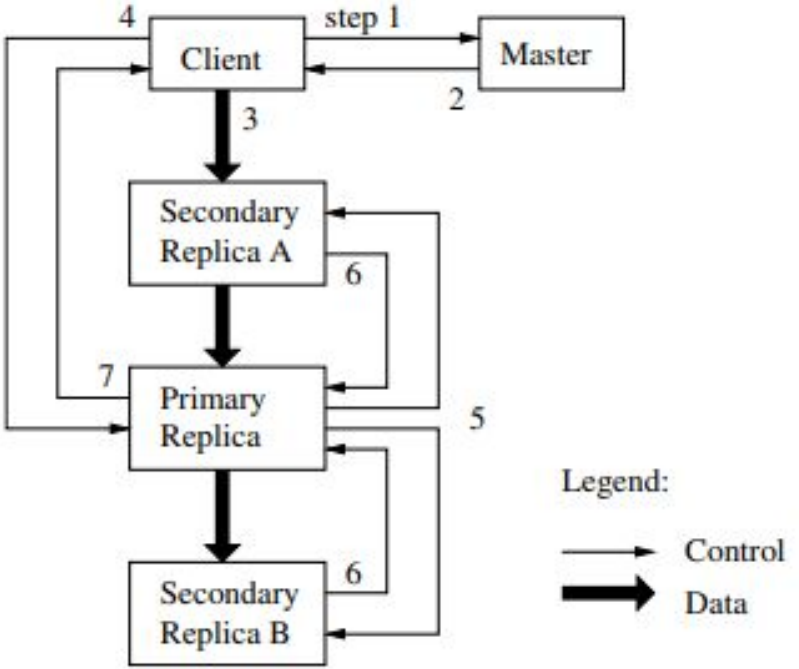
All metadata fits in master's memory

Metadata

Op log:

- persistently record metadata ops (e.g., create a file)
- replicated, persisted before ack, batch for amortizing cost
- checkpoint in-memory state, garbage collect log entries
- Q: why do this?

Writes



Data is replicated in a "chain" fashion

Let's say there are two writes to the same chunk

How to order?

Assign one as primary replica

Master gives a "lease" to primary

File Snapshot

Make a copy of a file

Use COW

First, revoke lease (can't modify)

Add create(F') to op log, point F' to same chunk

If a client wants to write, take a copy at that time

Copy on the same CS disk is faster than n/w

Flat namespace, Locking

There are NO inodes at the GFS-level!

On Unix/Linux: dir ops are serialized – create /foo/a.txt, create /foo/b.txt

GFS: take read locks on dirname, write lock on filenames allow concurrent creations within a single directory

Replica Management

Creation: balance disk space util, place across racks, limit recent create

Re-replication: when? failures, disk corruptions, errors etc

Rebalance: to spread load, disk space util

File Deletion and GC

When delete, just rename to hidden name

Immediately log the operation but lazily gc hidden files (after 3 days)

Remove name of hidden file

Identify orphan chunks -- those not pointed to by any file (similar to fsck)

Inform CS that free to drop those chunks

Simpler, neat approach than eager delete!

Data Integrity

Files stored on Linux, and Linux has bugs – sometimes silent corruptions

Files stored on disk, and disks are not fail-stop

- stored blocks can become corrupted over time
- rare events become common at scale

Cannot compare across replicas – legal to be divergent!

Chunkservers maintain per-chunk CRCs (64KB) – verify CRCs before returning read data

Record appends – incremental cksum, verify on read

Overwrites – read, verify, recompute, and write new cksum

Remarks

Radical departure from the norm

- but driven by needs
- largely successful

12 citations!

Cluster I/O NOW project at Berkeley

GFS □ Colossus

Assume GFS has 100 PB of data, how much metadata? – 10 TB metadata

Cannot store in one server's memory any more

Also,

- developers had to organize their apps around large append-only files
- latency-sensitive applications suffered

HDFS has same problem (single NameNode)

GFS eventually replaced with a new design, Colossus

Colossus supports much larger file systems