IOWA STATE UNIVERSITY
Department of Electrical and Computer Engineering

Lectures 37 - 39: Advanced Topics: Google File System & Beyond

1

Agenda

- Recap
- Advanced Topics
  - The Google File System (GFS)
  - The World After GFS

2

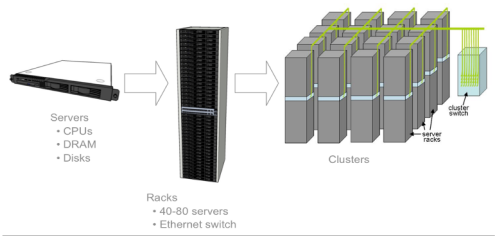
Recap

- What's cloud computing
  - Informal:
    - computing with large datacenters
    - users generally do not even know the exact location of "their" resources or even which country they are located in
  - NIST's five essential characteristics
- Types of cloud
  - Infrastructure as a Service (IaaS)
  - Platform as a Service (PaaS)
  - Software as a Service (SaaS)
  - Public vs. private clouds

amazon web services
Google Cloud
Microsoft Azure
chameleon
openstack
ARA

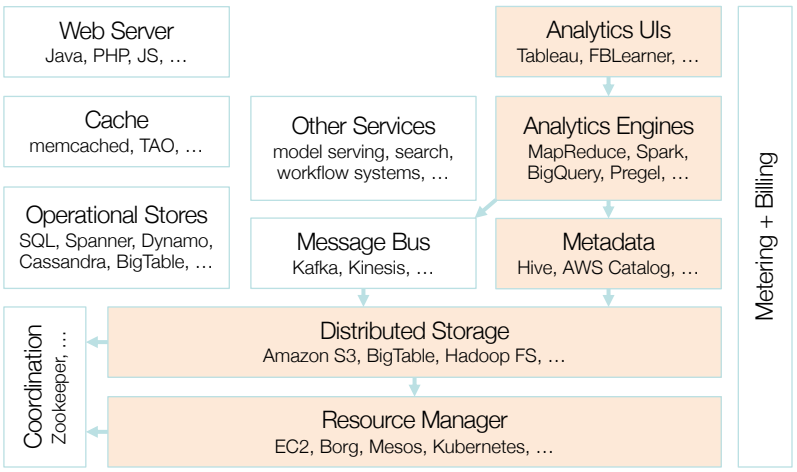
Recap

- Cloud Hardware
  - "Warehouse-Scale Computer" (WSC)
    - Datacenter as a computer
      - Rows of rack-mounted servers
      - Often organized as a few mostly independent clusters



Recap

- Cloud Software



5

Agenda

- ~~Recap~~
- Advanced Topics
  - The Google File System (GFS)
  - The World After GFS

6

Ordinary User's View of Google

- Search engine, Gmail, Google Maps, Ads, ...

CPRE 308/563X Students' View of Google

- A wide range of knowledge/techniques/problems/opportunities

Product design	…and much, much more!
User interfaces	
Machine learning, Statistics, Information retrieval, AI	
Data structures, Algorithms	
Compilers, Programming languages	
Networking, Distributed systems, Fault tolerance	
Hardware, Mechanical engineering	

Hardware Design Philosophy

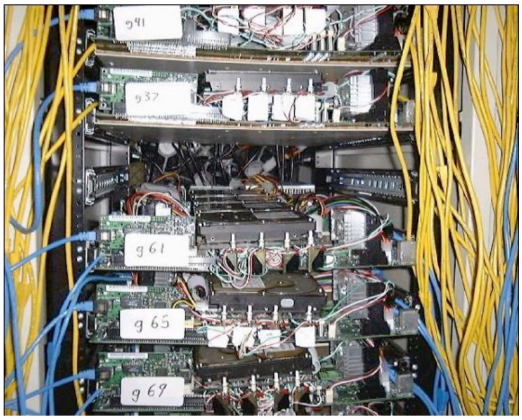
- "Scale out" vs. "Scale up"
  - Google prefers low-end server/PC-class designs
    - Build lots of them!
    - Why?
      - Single machine performance is not interesting
        - Even smaller problems at Google are too large for any single system
        - Large problems have lots of available parallelism
  - In contrast to the supercomputer designs in national labs
    - Highly customized/specialized

"Google" Circa 1997

- Started by 2 graduate students at Stanford University

Google Circa 1999

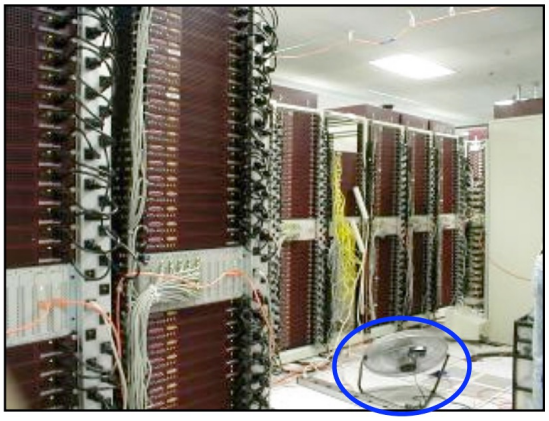
- Multiple blade servers



11

Google Circa 2000

- "Data center"



12

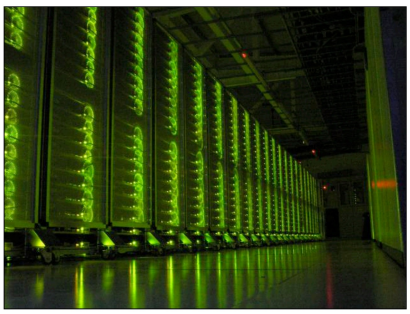
Google Circa 2001

- New Data Center
  - in three days ...

13

Google Today

- Multiple data centers around the world



14

The Joy of Real Hardware at Scale

- Typical first year for a new cluster:
  - ~0.5 overheating (power down most machines in< 5 min, ~1-2 days to recover)
  - ~20 rack failures (40-80 machines instantly disappear)
  - ~1 PDU failure (~500-1000 machines suddenly disappear)
  - ~5 racks go wonky (40-80 machines see 50% packet loss)
  - ~8 network maintenances (4 might cause connectivity losses)
  - ~1000 individual machine failures
  - ~thousands of hard drive failures
  - ~slow disks, bad memory, etc.
  - ...

- Long-haul networking breaks for unusual reasons, too
  - Wild dogs, dead horses, thieves, blasphemy, drunken hunters and sharks

15

Implications

- Stuff Breaks
  - If you have one server, it may stay up 1,000 days
  - If you have 10,000 servers, expect to lose ten a day
- "Ultra-reliable" hardware doesn't really help
  - At large scales, super-fancy reliable hardware still fails, albeit less often
    - software still needs to be fault-tolerant
    - commodity machines without fancy hardware give better perf/$

16

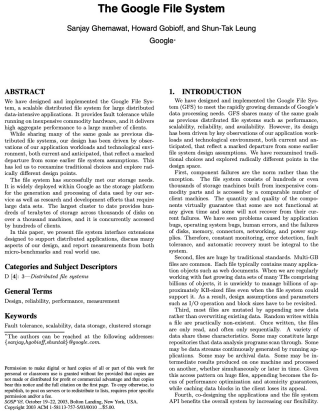
Implications

- Stuff Breaks
  - If you have one server, it may stay up 1,000 days
  - If you have 10,000 servers, expect to lose ten a day
- "Ultra-reliable" hardware doesn't really help
  - At large scales, super-fancy reliable hardware still fails, albeit less often
    - software still needs to be fault-tolerant
    - commodity machines without fancy hardware give better perf/$
- **"Failures are the norm rather than exception"**
  - Reliability has to come from the software

The Google File System

- One key infrastructure empowering Google
  - Published at the19th ACM Symposium on Operating Systems Principles (SOSP), 2003
  - https://research.google/pubs/pub51/
  - Driven by Google's applications/needs

Google Cloud



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {sanjay,hgobioff,shuntak}@google.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and that some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

The Google File System

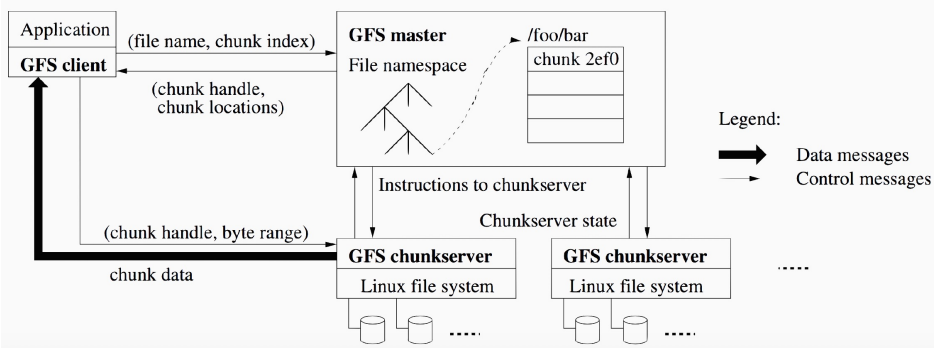
- Major Observations & Implications
  - Component failures are normal
    - due to both quantity & quality
    - need constant monitoring, error detection, fault tolerance, recovery etc.
  - Files are huge by traditional standards
    - many GBs/TBs
    - need to optimize for large I/O sizes
  - Most files are modified by appending at the end
    - Random writes (and overwrites) are practically non-existent
    - need to optimize for append operations
    - Caching data at client side becomes less attractive
  - Co-design applications and the file system benefits the overall system
    - increasing flexibility for both applications and the FS
      - E.g., may introduce new operation APIs

The Google File System

- Interface
  - A familiar file system interface
    - Files are organized hierarchically in directories & identified by path names
    - Does not fully follow POSIX standard
  - Support typical operations
    - *create, delete, open, close, read, write*
  - Introduce non-traditional operations
    - *snapshot*
      - Creates a copy of a file or a directory tree at low cost
    - *record append*
      - Allows multiple clients to append data to the same file concurrently
      - Guarantees atomicity of each individual client's append

The Google File System

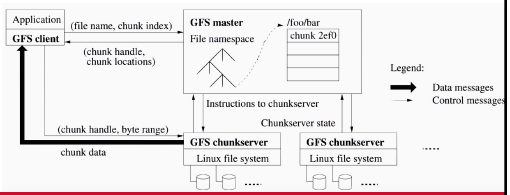
- Architecture
  - three types of nodes
    - master, chunkserver(s), client(s)
    - each is typically a Linux machine running a user-level server process



21

The Google File System

- Clients
  - Applications invoke GFS APIs
  - Ask the *master* which *chunkservers* to contact
    - i.e., metadata communication
  - Interact with *chunkservers* directly for reading/writing application data
    - i.e., data communication
  - Cache metadata
    - Why?
    - Why not data?



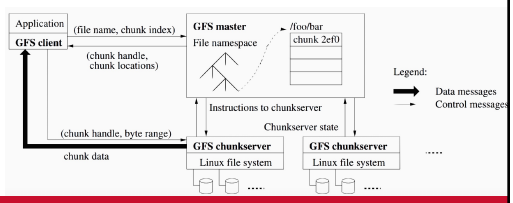
22

The Google File System

- Chunkservers
  - store all user files
    - Files are divided in fixed-size chunks
      - 64 MB (why?)
      - each is a Linux file on local file system (e.g., Ext4)
      - each is identified by a 64 bit unique handle in GFS
    - Do not cache file data (at GFS layer)
      - Linux buffer cache already put hot data in memory
  - replication
    - each chunk is replicated on multiple chunkservers
      - 3 by default (replication factor & locations are tunable)
    - tradeoff: space overhead vs. reliability/availability
      - erasure coding: achieve redundancy w/o full replication

The Google File System

- Master
  - Store all GFS metadata
    - file and chunk namespace
    - mapping from files to chunks
    - chunk locations (including replicas)
    - other info
      - E.g., access-control information
    - In-memory
  - Single master
    - why?



The Google File System

- Master
  - How to know chunk locations?
    - Chunkservers may “come and go”
      - Fail, reboot, change name, etc.
      - Hard to sync if keep locations persistently at master

25

The Google File System

- Master
  - How to know chunk locations?
    - Chunkservers may “come and go”
      - Fail, reboot, change name, etc.
      - Hard to sync if keep locations persistently at master
    - Solution
      - Poll chunkservers at startup
      - Keep up-to-date thereafter
        - Controls all chunk placement
        - Monitor chunkservers via periodic heartbeat messages

26

The Google File System

- Master
  - Maintain an operation log persistently
    - Central to GFS
      - A historical record of critical metadata changes
        - the only persistent record of metadata
      - Provide a logical time line that defines the order of all concurrent operations
      - Guarantee crash consistency at the GFS level
    - Store on local disk, replicate on multiple remote machines
      - Respond to a client operation only after flushing the corresponding log record to disk both locally and remotely

The Google File System

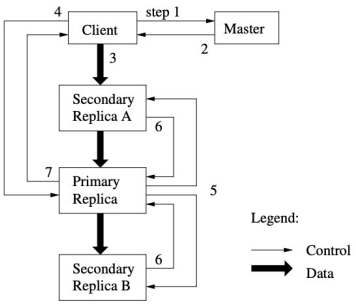
- Typical Workflow
  - Client
    - using fixed chunk size, translate filename & byte offset to chunk index, send request to master
  - Master
    - replies with chunk handle & location of chunkserver replicas (including which is “primary”)
  - Client
    - cache info using filename & chunk index as key
    - request data from nearest Chunkserver
    - no need to talk more about this 64MB chunk until cached info expires or file reopened
    - often initial request asks about sequence of chunks

The Google File System

- Lease and Mutation Order
  - Mutation
    - an operation that changes the contents or metadata of a chunk
    - E.g., write, append
  - Lease
    - A token for maintaining a consistent mutation order across replicas
  - Master grants a chunk "lease" to one replica
    - called "primary" chunkserver
    - the primary picks a serial order for all mutations to the chunk
      - serializes all mutation requests
      - communicates order to replicas

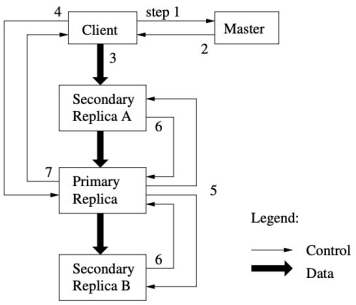
The Google File System

- Write Control and Data Flow
  - Step 1:
    - client asks the master which chunkserver holds the current lease for the chunk and the locations of the replicas
    - If no one has a lease, the master grants one to a replica



The Google File System

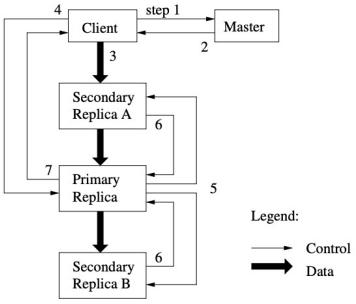
- Write Control and Data Flow
  - Step 2:
    - Master replies with the ID of the primary and secondary replicas
    - Client cache the info for future mutations



31

The Google File System

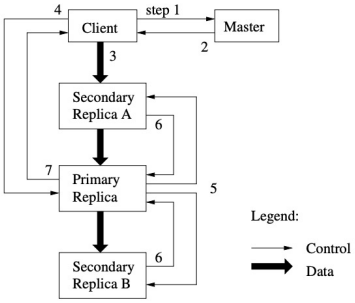
- Write Control and Data Flow
  - Step 3:
    - Client pushes the data to all the replicas
      - Order does not matter
    - Each chunkserver store the data in an LRU buffer in memory
      - Acknowledge to client after receiving the data



32

The Google File System

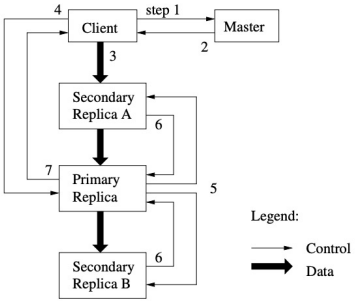
- Write Control and Data Flow
  - Step 4:
    - After all replicas have acknowledged receiving the data, the client sends a write request to the primary
    - The primary assigns consecutive serial numbers to all the mutations it receives, and applies the mutation to its own local state in order



33

The Google File System

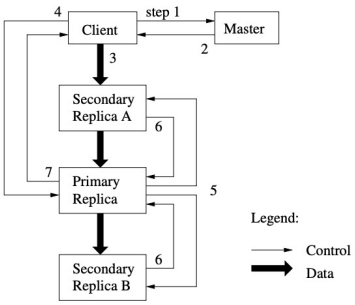
- Write Control and Data Flow
  - Step 5:
    - The primary forwards the write request to all other replicas
      - including the serial numbers
    - Each secondary replica applies mutations in the same serial number order assigned by the primary



34

The Google File System

- Write Control and Data Flow
  - Step 6:
    - All secondary replicas reply to the primary indicating that they have completed the operation
  - Step 7:
    - The primary replies to the client
      - Report errors if occurred
      - Client may retry in case errors



35

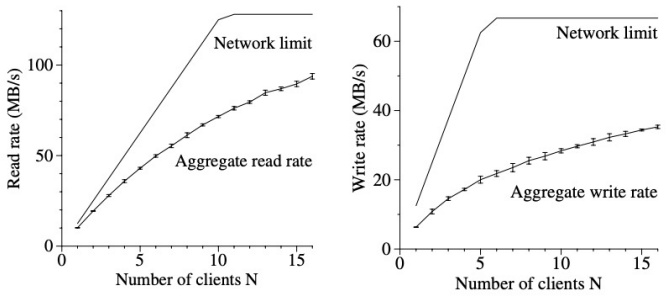
The Google File System

- Other issues
  - Atomic Appends
  - Fast snapshot
  - Master operation
    - namespace management & locking
    - replica placement & rebalancing
    - garbage collection (deleted/stale files)
    - detecting stale replicas
  - Master replication
    - master log & checkpoints replicated
    - shadow masters
      - provide read-access when primary is down

36

The Google File System

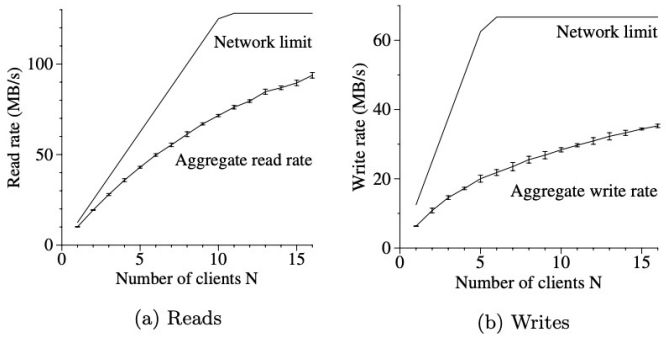
- Read & Write Performance
  - Both are scalable
    - i.e., read/write rates increase as the client count increases



37

The Google File System

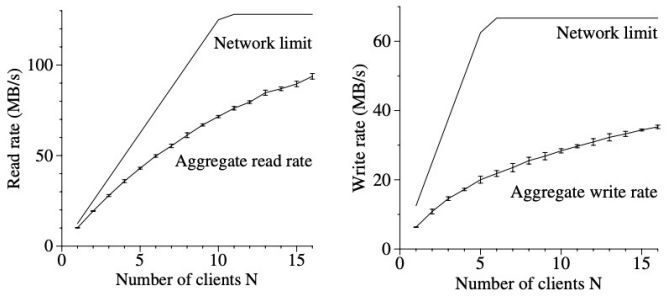
- Read & Write Performance
  - Why does the increasing rate drop?



38

The Google File System

- Read & Write Performance
  - Why does the increasing rate of writes drop more?



39

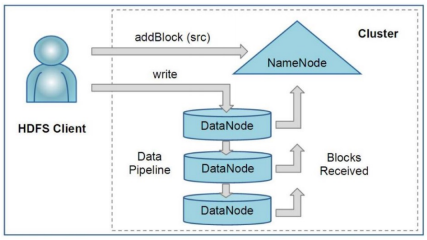
Agenda

- ~~Recap~~
- Advanced Topics
  - ~~The Google File System (GFS)~~
  - The World After GFS

40

Hadoop Distributed File System (HDFS)

- An open source version of GFS
  - formally published at *26th IEEE Symposium on Massive Storage Systems and Technologies (MSST'10)*
  - NameNode + DataNodes
  - topology-aware replica



The Hadoop Distributed File System

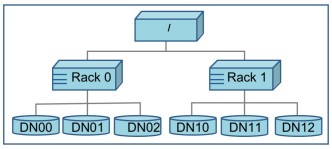
Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler
Yahoo!
Sunnyvale, California USA
{Shv, Hairong, SRadia, Chansler}@Yahoo-Inc.com

Abstract—**The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size. We describe the architecture of HDFS and report on experience using HDFS to manage 25 petabytes of enterprise data at Yahoo!.**

developed at Facebook. Pig [4], ZooKeeper [6], and Chukwa were originated and developed at Yahoo! Avro was originated at Yahoo! and is being co-developed with Cloudera.

HDFS is the file system component of Hadoop. While the interface to HDFS is patterned after the UNIX file system, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand.

HDFS stores file system metadata and application data



41

Hadoop Distributed File System (HDFS)

- Enable many follow-up research/innovations
  - across academia and industry

Analysis of HDFS Under HBase: A Facebook Messages Case Study

Tyler Harter, Dhruba Borthakur[†], Siying Dong[†], Amitanand Aiyer[†],
Liyin Tang[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison [†] Facebook Inc.

Abstract

We present a multilayer study of the Facebook Messages stack, which is based on HBase and HDFS. We collect and analyze HDFS traces to identify potential improvements, which we then evaluate via simulation. Messages represents a new HDFS workload: whereas HDFS

ingly common, layered storage architecture: a distributed database (HBase, derived from BigTable [3]) atop a distributed file system (HDFS [24], derived from the Google File System [11]). Our goal is to study the interaction of these important systems, with a particular focus on the lower layer; thus, our highest-level question: is HDFS an effective storage backend for HBase?

A Tale of Two Erasure Codes in HDFS

Mingyuan Xia[*], Mohit Saxena[†], Mario Blaum[†] and David A. Pease[†]
[*]*McGill University,* [†]*IBM Research Almaden*
mingyuan.xia@mail.mcgill.ca,{msaxena,mblaum,pease}@us.ibm.com

Abstract

Distributed storage systems are increasingly transitioning to the use of erasure codes since they offer higher reliability at significantly lower storage costs than data replication. However, these codes tradeoff recovery performance as they require multiple disk reads and network

replicated across multiple machines and racks. For example, the Google File System [11] and the Hadoop Distributed File System [4] maintain three copies of each data block. Although disk storage seems inexpensive today, replication of the entire data footprint is simply infeasible at massive scales of operation. As a result, most

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling
KTH - Royal Institute of Technology
{smkniazi, maism, haridi, jdowling}@kth.se

Steffen Grohsschmiedt
Spotify AB
steffeng@spotify.com

Mikael Ronström
Oracle
mikael.ronstrom@oracle.com

Abstract

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL databases have reopened the research question of whether distributed metadata for hierarchical file systems can be managed using commodity databases. In this paper, we introduce HopsFS, a next generation distribution of the Hadoop Distributed File System (HDFS) that replaces HDFS' single node in-memory metadata service,

shards that change in size over time.

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL [42] databases have reopened the possibility of storing distributed file system metadata in a commodity database. To date, the conventional wisdom has been that it is too expensive (in terms of throughput and latency) to store hierarchical file system metadata fully normalized in a distributed database [59, 33].

HARDFS: Hardening HDFS with Selective and Lightweight Versioning

Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison [†] University of Chicago

Abstract

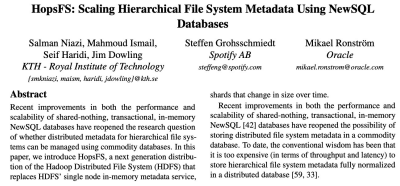
We harden the Hadoop Distributed File System (HDFS) against fail-silent (non-fail-stop) behaviors that result from memory corruption and software bugs using a new approach: selective and lightweight versioning (SLEEVE). With this approach, actions performed by

a single point of failure in today's systems. Observations from real systems show that these failures can lead to transient, non-deterministic errors, and make the system exhibit *fail-silent* behaviors (*e.g.*, send corrupt messages) rather than crashing; these fail-silent errors can lead to data loss, unavailability, and prolonged debug-

42

Hadoop Distributed File System (HDFS)

- Enable many follow-up research/innovations
  - E.g., HopsFS (FAST'17)
    - solve HDFS namenode's scalability issue
      - Limited metadata namespace
      - Limited concurrency
    - Leverage MySQL NDB Cluster to scale out metadata
    - 16X-37X the throughput of HDFS
    - Open source



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling
KTH - Royal Institute of Technology
{smkniazi, maism, haridi, jdowling}@kth.se

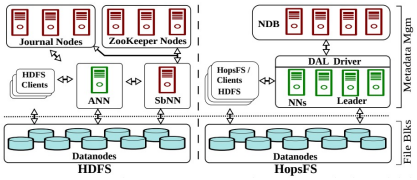
Steffen Grohsschmiedt
Spotify AB
steffeng@spotify.com

Mikael Ronström
Oracle
mikael.ronstrom@oracle.com

Abstract

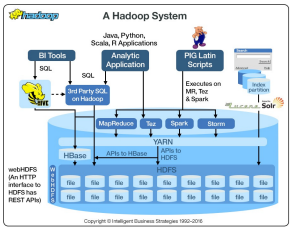
Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL databases have reopened the research question of whether distributed metadata for hierarchical file systems can be managed using commodity databases. In this paper, we introduce HopsFS, a next generation distribution of the Hadoop Distributed File System (HDFS) that replaces HDFS' single node in-memory metadata service, shards that change in size over time.

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL [42] databases have reopened the possibility of storing distributed file system metadata in a commodity database. To date, the conventional wisdom has been that it is too expensive (in terms of throughput and latency) to store hierarchical file system metadata fully normalized in a distributed database [59, 33].



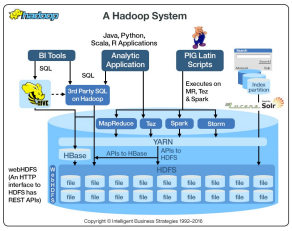
Hadoop Ecosystem

- Many software frameworks built atop HDFS
  - MapReduce, Spark, Spark Streaming, Hive, etc.
    - processing of live data streams, SQL queries on big data, etc.
      - developed by Apache Software Foundation
        - major vendors include Cloudrea, Hortonworks, MapR Tech, Dell/EMC/Pivotal, IBM, Microsoft, AWS, etc.
        - market size expected to reach $ 8240.1 million by 2026

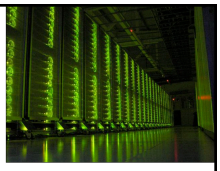


Hadoop Ecosystem

- Many software frameworks built atop HDFS
  - MapReduce, Spark, Spark Streaming, Hive, etc.
    - processing of live data streams, SQL queries on big data, etc.
      - developed by Apache Software Foundation
        - major vendors include Cloudrea, Hortonworks, MapR Tech, Dell/EMC/Pivotal, IBM, Microsoft, AWS, etc.
        - market size expected to reach $ 8240.1 million by 2026

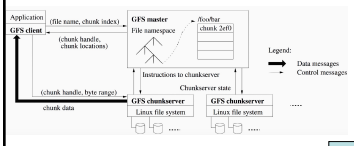


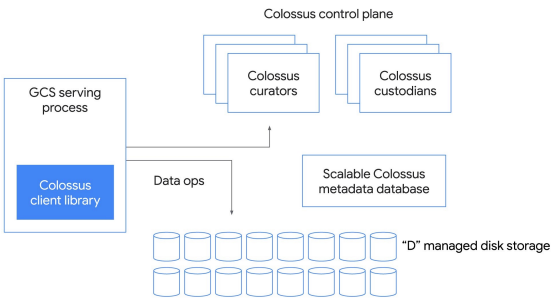
45



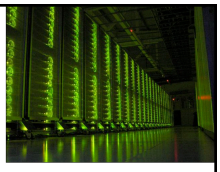
How about GFS itself?

- Colossus: successor of GFS
  - Google's cluster-level file system today
  - Enhanced storage scalability and availability
    - to handle the massive growth in data needs of an ever-growing number of applications



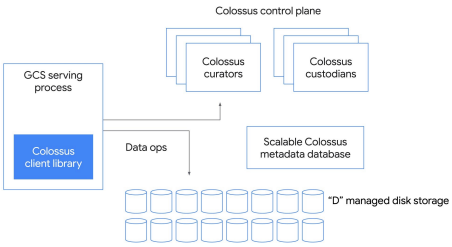


46



How about GFS itself?

- Colossus: successor of GFS
  - Client Library
    - Includes a lot of functionality (e.g., software RAID)
    - Tunable based on application needs
  - Control Plane
    - Curators: handle client requests (e.g., file creation)
    - Metadata database
      - Store metadata in Google's NoSQL database BigTable
      - Scale up by over 100x over GFS
    - Custodians
      - background maintainance (e.g., disk space balancing, RAID reconstruction)



47

Building Blocks of Google Storage

- Colossus: successor of GFS
- Spanner: scalable relational database (OSDI'12)
- Borg: cluster manager (EuroSys'15)

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: nonblocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

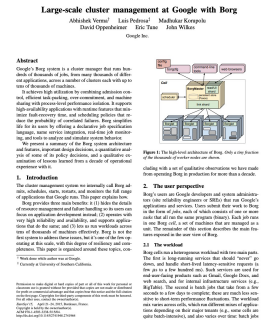
Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spanning the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of the distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by the authors of [37].) As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

Published in the Proceedings of OSDI 2012



Large-scale cluster management at Google with Borg

Abhishek Verma† Luis Pedrosa‡ Madhukar Korupolu David Oppenheimer Eric Tune John Wilkes

Google Inc.

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it's one of the few operating at this scale, with this degree of resilience and completeness. This paper is organized around these topics, con-

† Work done while author was at Google.

‡ University of Edinburgh, Scotland.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

EuroSys'15, April 21–24, 2015, Bordeaux, France.

ACM 978-1-4503-3238-5/15/04.

http://dx.doi.org/10.1145/2741948.2741964

1. Overview

2. The user perspective

Borg's users are Google developers and system administrators (site reliability engineers or SREs) that run Google's applications and services. Users submit their work to Borg in the form of jobs, each of which consists of one or more tasks all running the same program (binary). Each task maps to one Borg cell, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

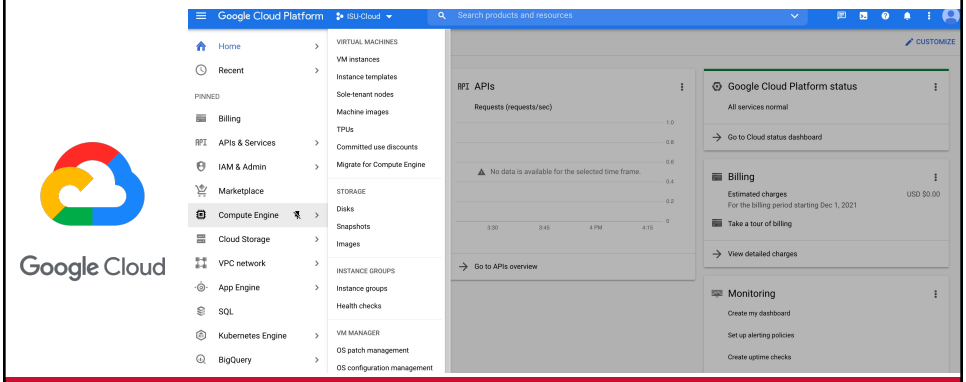
2.1 The workload

Borg cells run a heterogeneous workload with two main parts. The first part, long-running services that should "never" go down, and handle short-lived latency-sensitive requests (a few µs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second part, batch jobs that take from seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

48

Fundamental to Google Cloud

- Various services enabled
  - E.g., VM, container, serverless functions, storage, database, …
    - Conceptually similar to ChameleonCloud used in bonus assignment, but much more diverse
  - Free trial ($300 credits)!



Google Cloud

Agenda

- ~~Recap~~
- ~~Advanced Topics~~
  - ~~The Google File System (GFS)~~
  - ~~The World After GFS~~

Questions?

*acknowledgement: slides include content adapted from “Modern Operating Systems” by A. Tanenbaum, “Operating Systems Concepts” by A. Silberschatz etc., “Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau etc., lectures from Google, Stanford, UC Berkeley, and anonymous pictures from internet