Cloud Scale Storage Systems

Yunhao Zhang
& Matthew Gharrity

Two Beautiful Papers

- Google File System
  - SIGOPS Hall of Fame!
  - pioneer of large-scale storage system
- Spanner
  - OSDI’12 Best Paper Award!
  - Big Table got SIGOPS Hall of Fame!
  - pioneer of globally consistent database

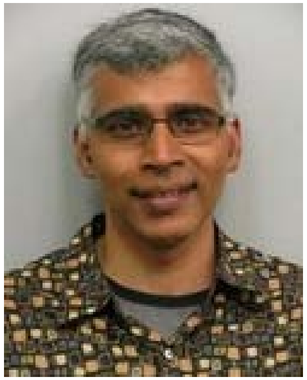
Topics in Distributed Systems

- GFS
  - Fault Tolerance
  - Consistency
  - Performance & Fairness
- Spanner
  - Clock (synchronous v.s. asynchronous)
  - Geo-replication (Paxos)
  - Concurrency Control

Google File System

Rethinking Distributed File System Tailored for the Workload

Authors



Sanjay Ghemawat

Cornell->MIT->Google

Howard Gobioff

R.I.P.

Shun-tak Leung

UW->DEC->Google

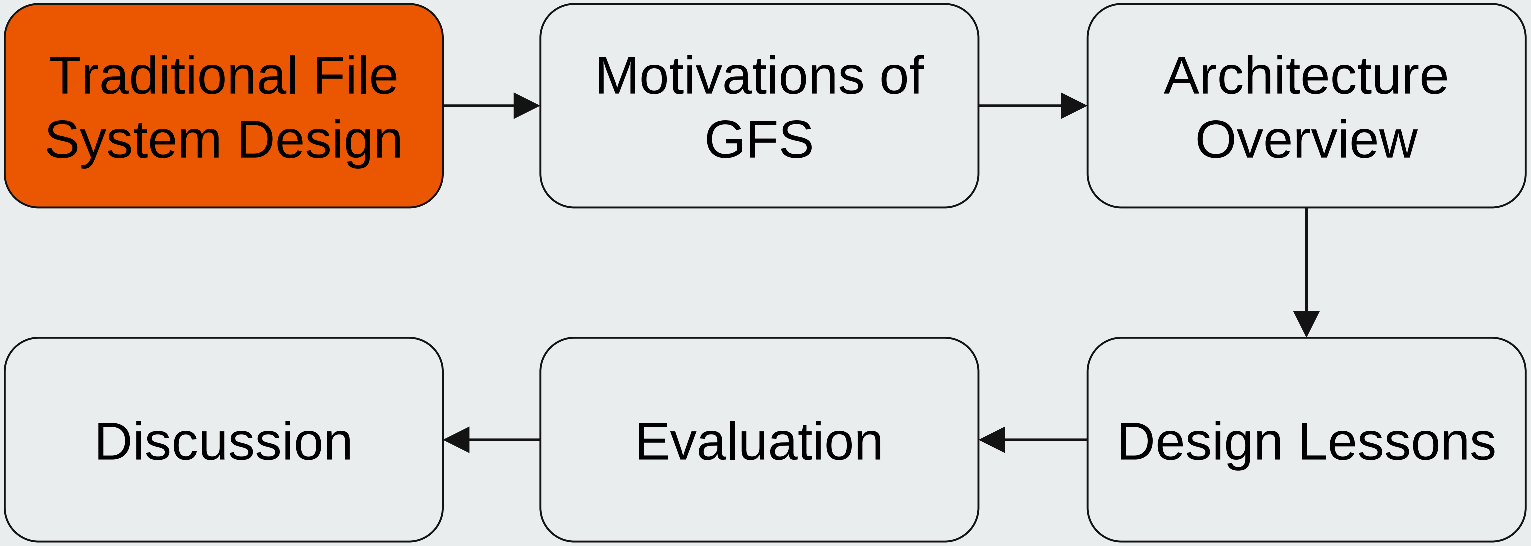
Evolution of Storage System (~2003)

- P2P routing/DistributedHashTables (Chord, CAN, etc.)
- P2P storage (Pond, Antiquity)
    - data stored by decentralized strangers
- cloud storage
    - centralized data center network at Google
- Question: Why using centralized data centers?

Evolution of Storage System (~2003)

- benefits of data center
  - centralized control, one administrative domain
  - seemingly infinite resources
  - high network bandwidth
  - availability
  - building data center with commodity machines is easy

Roadmap


Traditional File System Design
Motivations of GFS
Architecture Overview
Design Lessons
Evaluation
Discussion

Recall UNIX File System Layers

Table 2–2: The naming layers of Unix.

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	
Absolute path name layer	Provide a root for the naming hierarchies.	user-oriented names
Path name layer	Organize files into naming hierarchies.	
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	
File layer	Organize blocks into files.	machine-oriented names
Block layer	Identify disk blocks.	

high level functionalities

filenames and directories

machine-oriented file id

disk blocks

Table borrowed from “Principles of Computer System Design” by J.H. Saltzer

Recall UNIX File System Layers

Table 2–2: The naming layers of Unix.

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	user-oriented names
Absolute path name layer	Provide a root for the naming hierarchies.	
Path name layer	Organize files into naming hierarchies.	
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	machine-oriented names
File layer	Organize blocks into files.	
Block layer	Identify disk blocks.	

Question: How GFS move from traditional file system design?

In GFS, what layers disappear?

What layers are managed by the master?

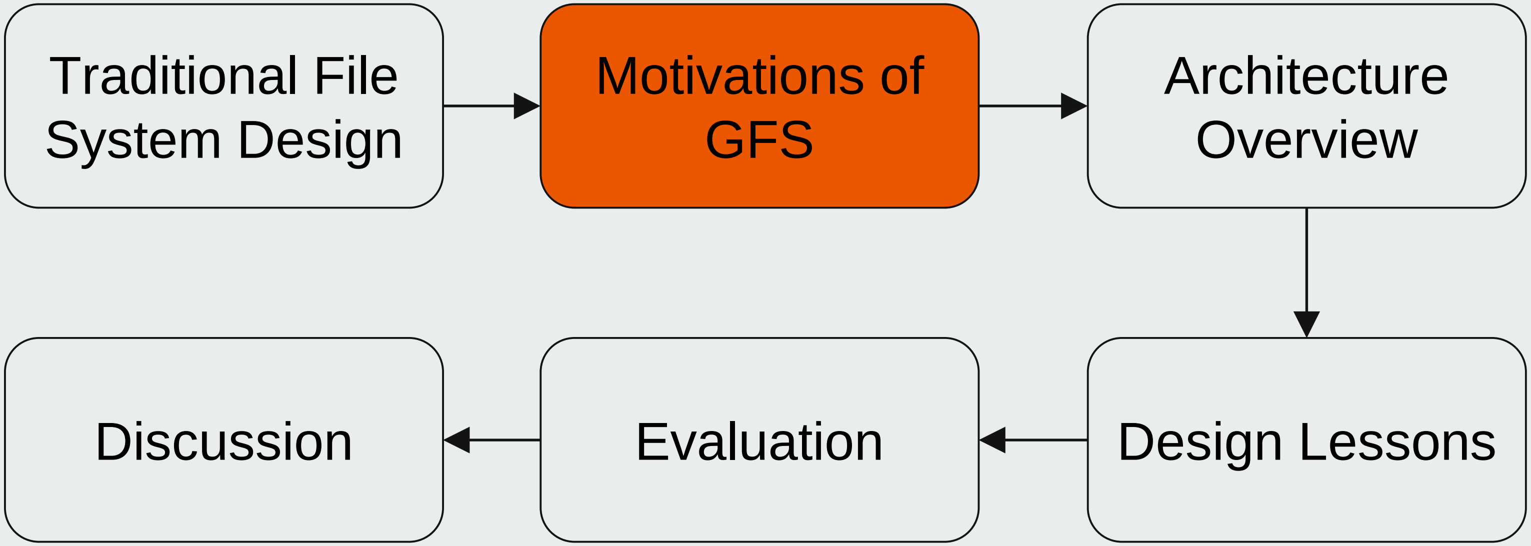
What are managed by the chunkserver?

Table borrowed from “Principles of Computer System Design” by J.H. Saltzer

Recall NFS

- distributed file system
- assume same access pattern of UNIX FS (transparent)
- no replication: any machine can be client or server
- stateless: no lock
- cache: files cache for 3 sec, directories cache for 30 sec
- problems
  - inconsistency may happen
  - append can't always work
  - assume clocks are synchronized
  - no reference counter

Roadmap


Traditional File System Design
Motivations of GFS
Architecture Overview
Design Lessons
Evaluation
Discussion

Different Assumptions

1. inexpensive commodity hardware
2. failures are norm rather than exception
3. large file size (multi-GB, 2003)
4. large sequential read/write & small random read
5. concurrent append
6. codesigning applications with file system

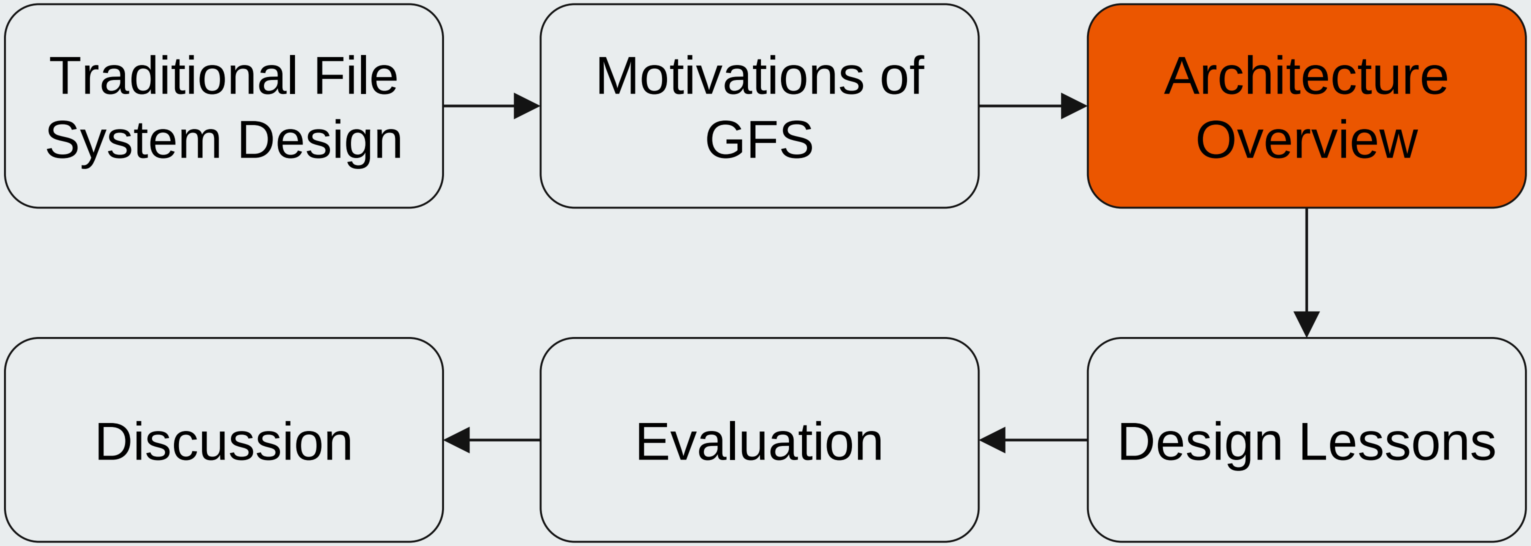
A Lot of Questions Marks on My Head

1. inexpensive commodity hardware (why?)
2. failures are norm rather than exception (why?)
3. large file size (multi-GB, 2003) (why?)
4. large sequential read/write & small random read (why?)
5. concurrent append (why?)
6. codesigning applications with file system (why?)

So, why?

1. inexpensive commodity hardware (why?)
    a. cheap! (poor)
    b. have they abandoned commodity hardware? why?
2. failures are norm rather than exception (why?)
    a. too many machines!
3. large file size (multi-GB, 2003) (why?)
    a. too much data!
4. large sequential read/write & small random read (why?)
    a. throughput-oriented v.s. latency-oriented
5. concurrent append (why?)
    a. producer/consumer model
6. codesigning applications with file system (why?)
    a. customized fail model, better performance, etc.

Roadmap


Traditional File System Design
Motivations of GFS
Architecture Overview
Design Lessons
Evaluation
Discussion

Moving to Distributed Design

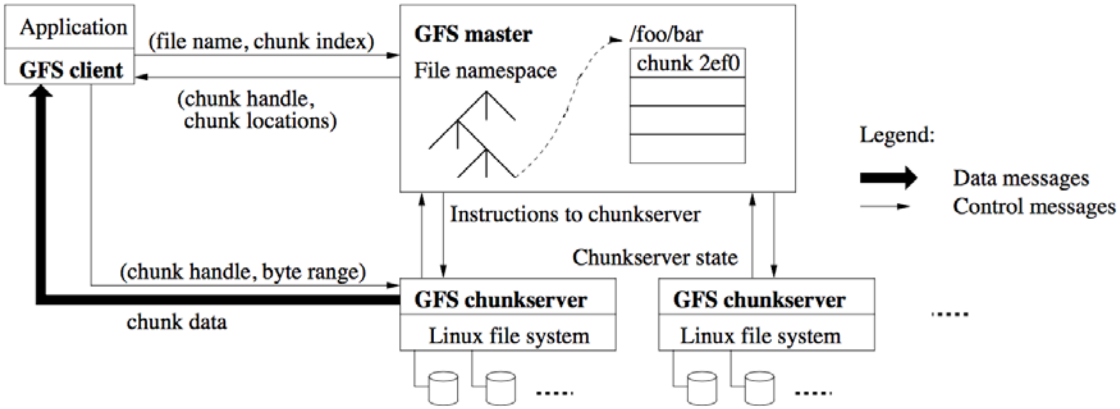


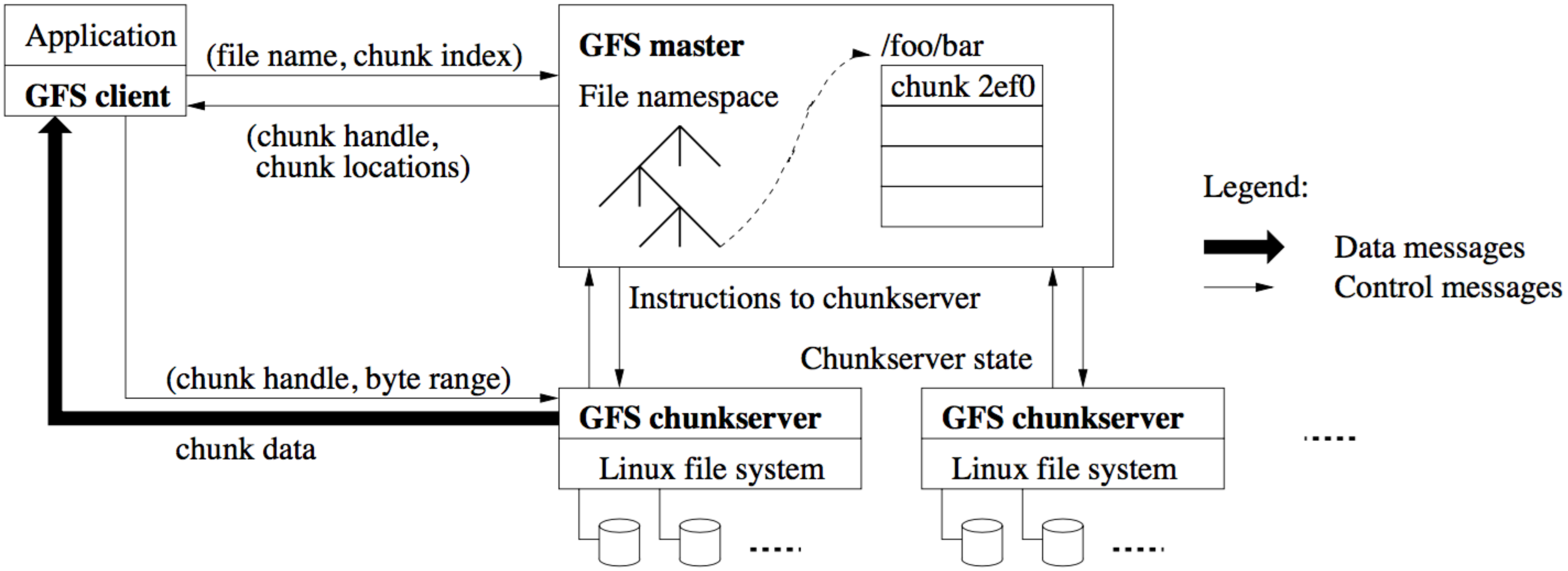
Table 2–2: The naming layers of Unix.

Layer	Purpose
Symbolic link layer	Integrate multiple file systems with symbolic links.
Absolute path name layer	Provide a root for the naming hierarchies.
Path name layer	Organize files into naming hierarchies.
File name layer	Provide human-oriented names for files.
Inode number layer	Provide machine-oriented names for files.
File layer	Organize blocks into files.
Block layer	Identify disk blocks.

Architecture Overview

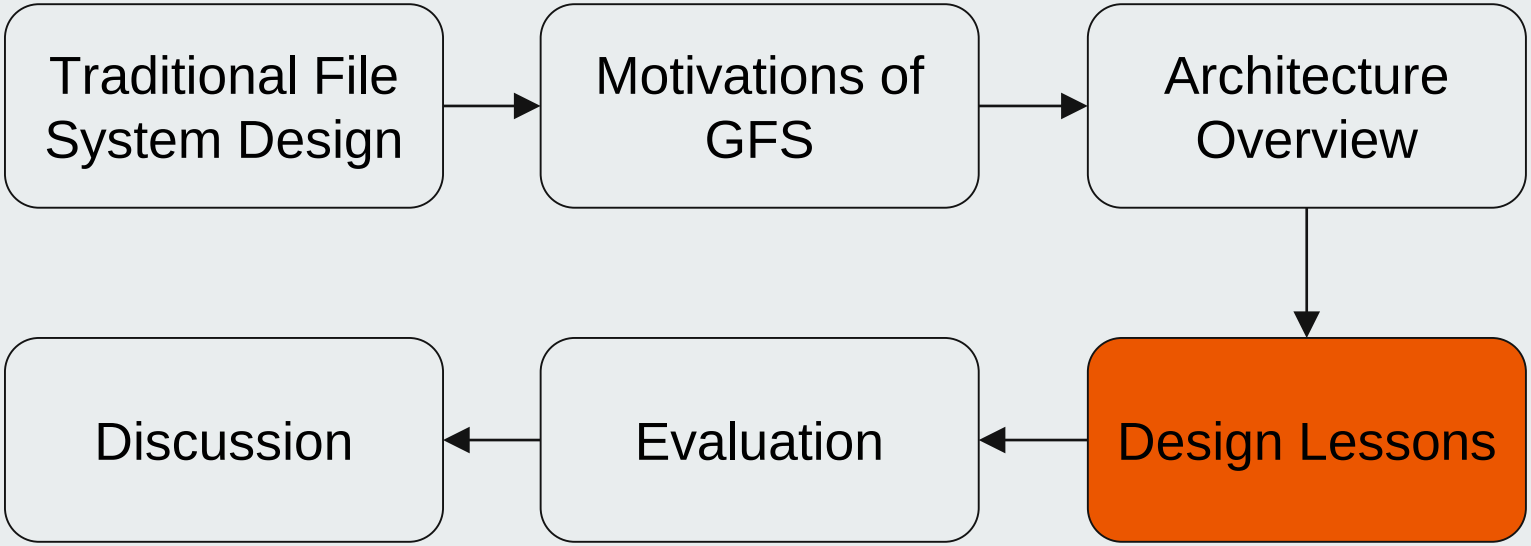
- GFS Cluster (server/client)
  - single master + multiple chunkservers
- Chunkserver
  - fixed sized chunks (64MB)
  - each chunk has a globally unique 64bit chunk handle
- Master
  - maintains file system metadata
    - namespace
    - access control information
    - mapping from files to chunks
    - current locations of chunks
  - Question: what to be made persistent in operation log? Why?

Architecture Overview



Discussion Question: Why using Linux file system? Recall Stonebraker’s argument.

Roadmap



Traditional File System Design
Motivations of GFS
Architecture Overview
Design Lessons
Evaluation
Discussion

Major Trade-offs in Distributed Systems

- Fault Tolerance
- Consistency
- Performance
- Fairness

Recall Assumptions

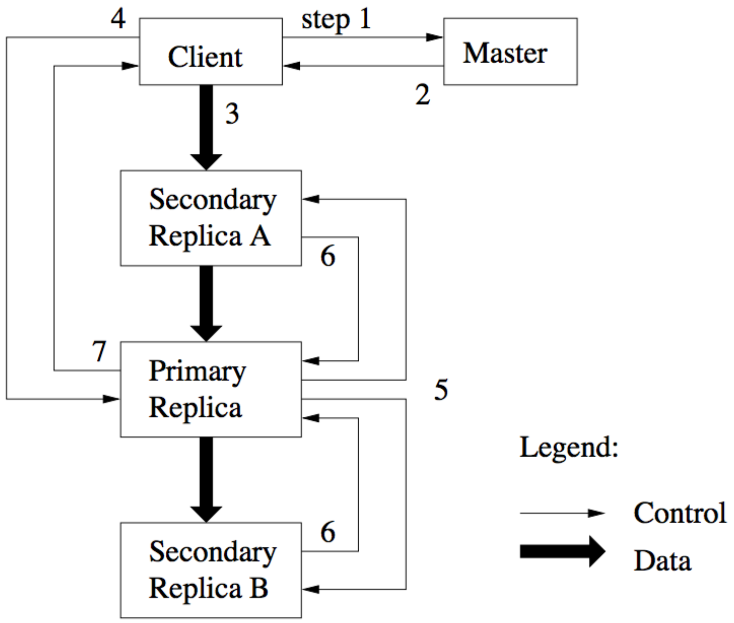
1. inexpensive commodity hardware
2. failures are norm rather than exception
3. large file size (multi-GB, 2003)
4. large sequential read/write & small random read
5. concurrent append
6. codesigning applications with file system

What is Fault Tolerance?

- fault tolerance is the art to keep breathing while dying
- before we start, some terminologies
  - error, fault, failure
    - why not error tolerance or failure tolerance?
  - crash failure v.s. fail-stop
    - which one is more common?

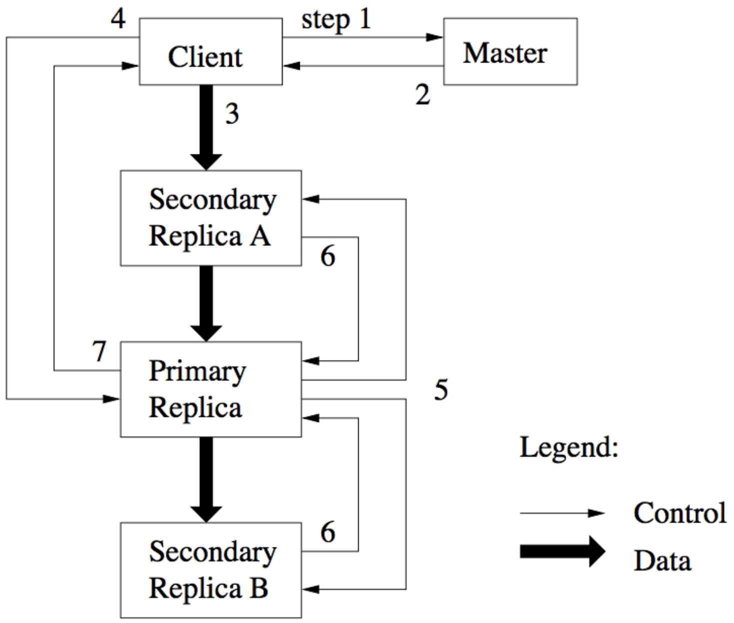
Fault Tolerance: Keep Breathing While Dying

- GFS design practice
  - primary / backup
  - hot backup v.s. cold backup



Fault Tolerance: Keep Breathing While Dying

- GFS design practice
  - primary / backup
  - hot backup v.s. cold backup
- two common strategies:
  - logging
    - master operation log
  - replication
    - shadow master
    - 3 replica of data
  - Question: what’s the difference?



My Own Understanding

- logging
  - **atomicity + durability**
  - on persistent storage (potentially slow)
  - little space overhead (with checkpoints)
  - asynchronous logging: good practice!
- replication
  - **availability + durability**
  - in memory (fast)
  - double / triple space needed
  - Question: How can (shadow) masters be inconsistent?

Major Trade-offs in Distributed Systems

- Fault Tolerance
  - logging + replication
- Consistency
- Performance
- Fairness

What is Inconsistency?

inconsistency!

client is angry!

How can we save the young man’s life?

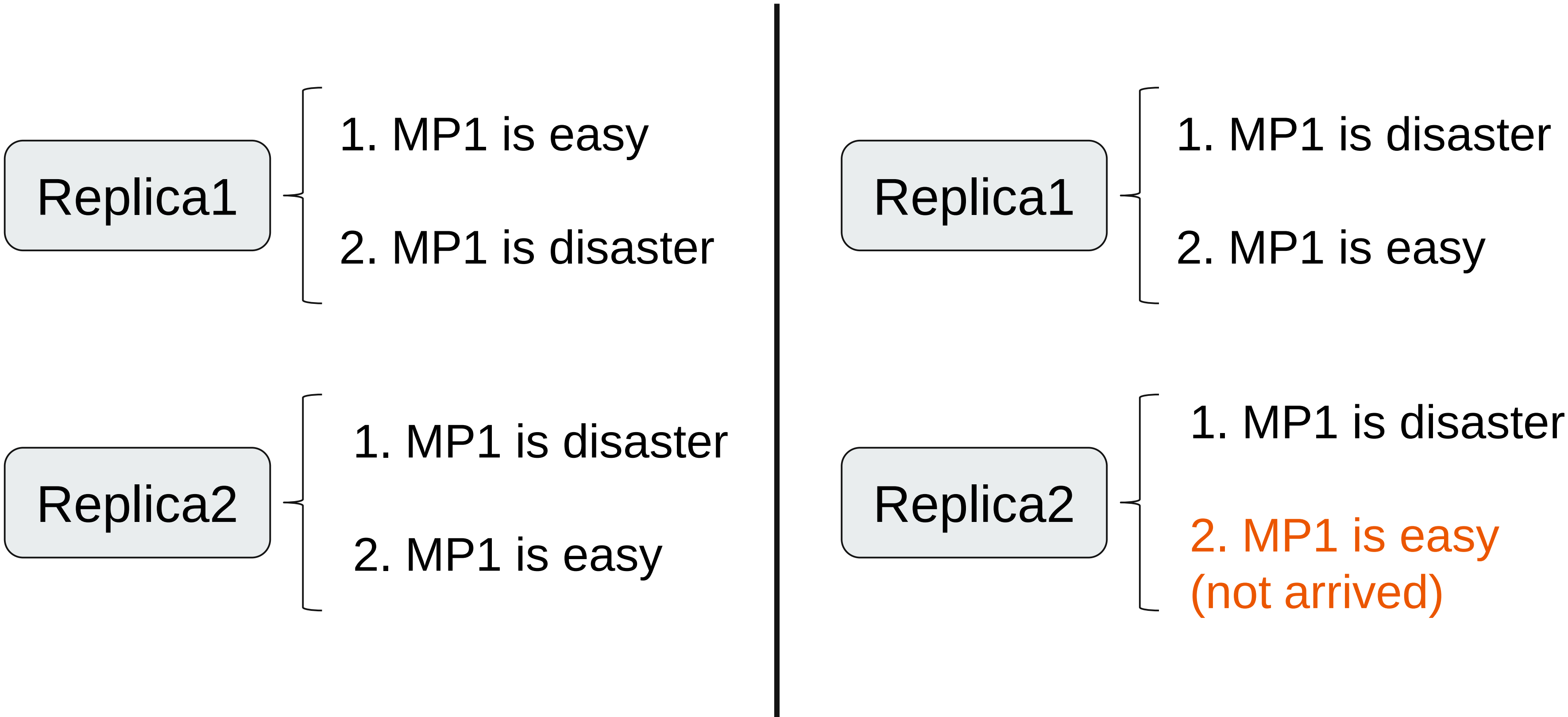
- Question: What is consistency? What cause inconsistency?

How can we save the young man’s life?

- Question: What is consistency? What cause inconsistency?
- Consistency model defines rules for the apparent order and visibility of updates (mutation), and it is a continuum with tradeoffs.

-- Todd Lipcon

Causes of Inconsistency



Order **Visibility**

Avoid Inconsistency in GFS

1. inexpensive commodity hardware
2. failures are norm rather than exception
3. large file size (multi-GB, 2003)
4. large sequential read/write & small random read
5. concurrent append
6. codesigning applications with file system

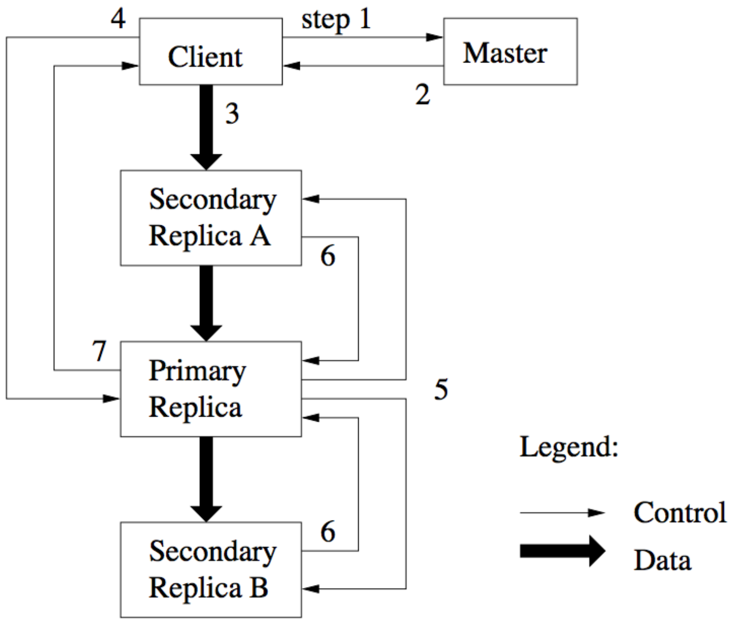
Mutation → Consistency Problem

- mutations in GFS
  - write
  - record append
- consistency model
  - defined (atomic)
  - consistent
  - optimistic mechanism v.s. pessimistic mechanism (why?)

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

Mechanisms for Consistent Write & Append

- Order: lease to primary and primary decides the order
- Visibility: version number eliminates stale replicas
- Integrity: checksum



Consistency model defines rules for the apparent order and visibility of updates (mutation), and it is a continuum with tradeoffs. -- Todd Lipcon

However, clients cache chunk locations!

- Recall NFS
- Question: What’s the consequence? And why?

Major Trade-offs in Distributed Systems

- Fault Tolerance
  - logging + replication
- Consistency
  - mutation order + visibility == lifesaver!
- Performance
- Fairness

Recall Assumptions

1. inexpensive commodity hardware
2. failures are norm rather than exception
3. large file size (multi-GB, 2003)
4. large sequential read/write & small random read
5. concurrent append
6. codesigning applications with file system

Performance & Fairness

- principle: avoid bottle-neck! (recall Amdahl's Law)

Performance & Fairness

- principle: avoid bottle-neck! (recall Amdahl's Law)
- minimize the involvement of master
    - client cache metadata
    - lease authorize the primary chunkserver to decide operation order
    - namespace management allows concurrent mutations in same directory

Performance & Fairness

- principle: avoid bottle-neck! (recall Amdahl’s Law)
- minimize the involvement of master
- chunkserver may also be bottle-neck
  - split data-flow and control-flow
  - pipelining in data-flow
  - data balancing and re-balancing
  - operation balancing by indication of recent creation

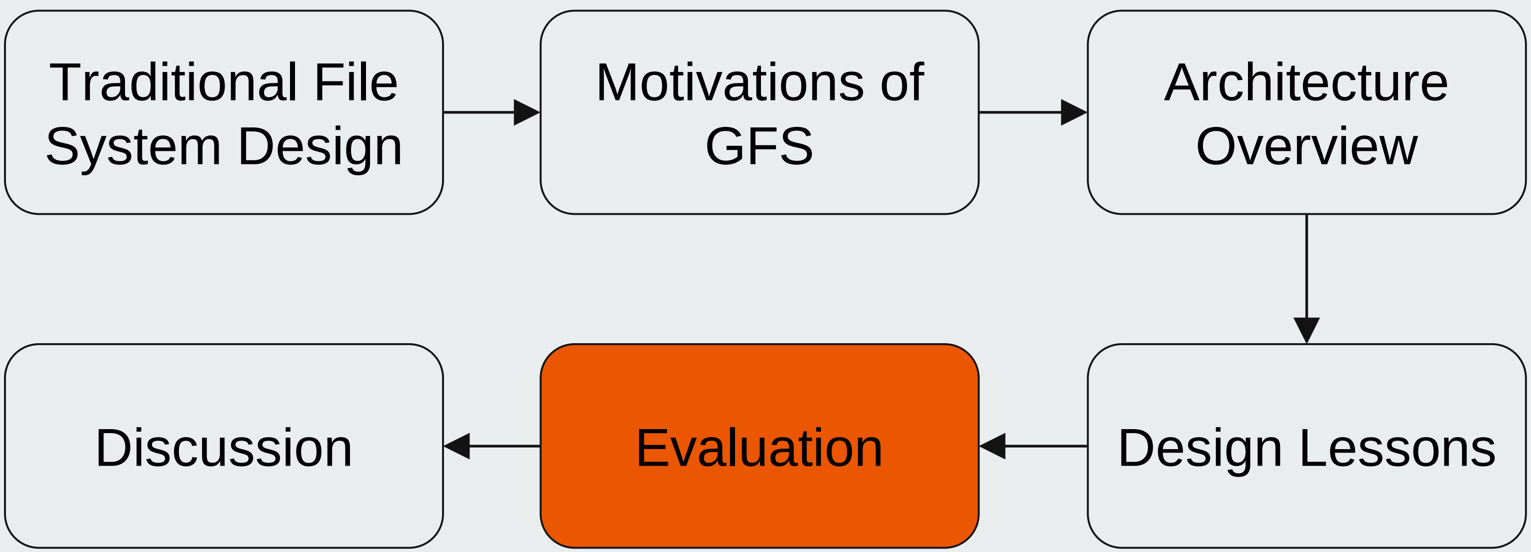
Performance & Fairness

- principle: avoid bottle-neck! (recall Amdahl’s Law)
- minimize the involvement of master
- chunkserver may also be bottle-neck
- time-consuming operations
  - make garbage collection in background

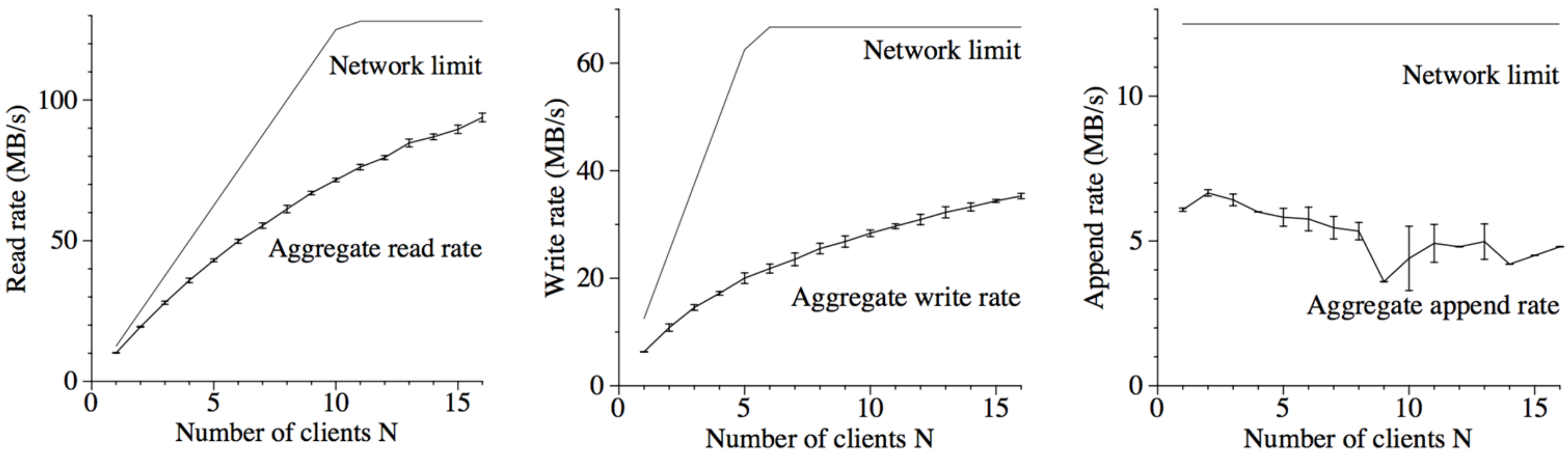
Conclude Design Lessons

- Fault Tolerance
  - logging + replication
- Consistency
  - mutation order + visibility == lifesaver!
- Performance
  - locality!
  - work split enables more concurrency
  - fairness work split maximize resource utilization
- Fairness
  - balance data & balance operation

Roadmap


Traditional File System Design
Motivations of GFS
Architecture Overview
Design Lessons
Evaluation
Discussion

Throughput



(a) Reads (b) Writes (c) Record appends

Breakdown

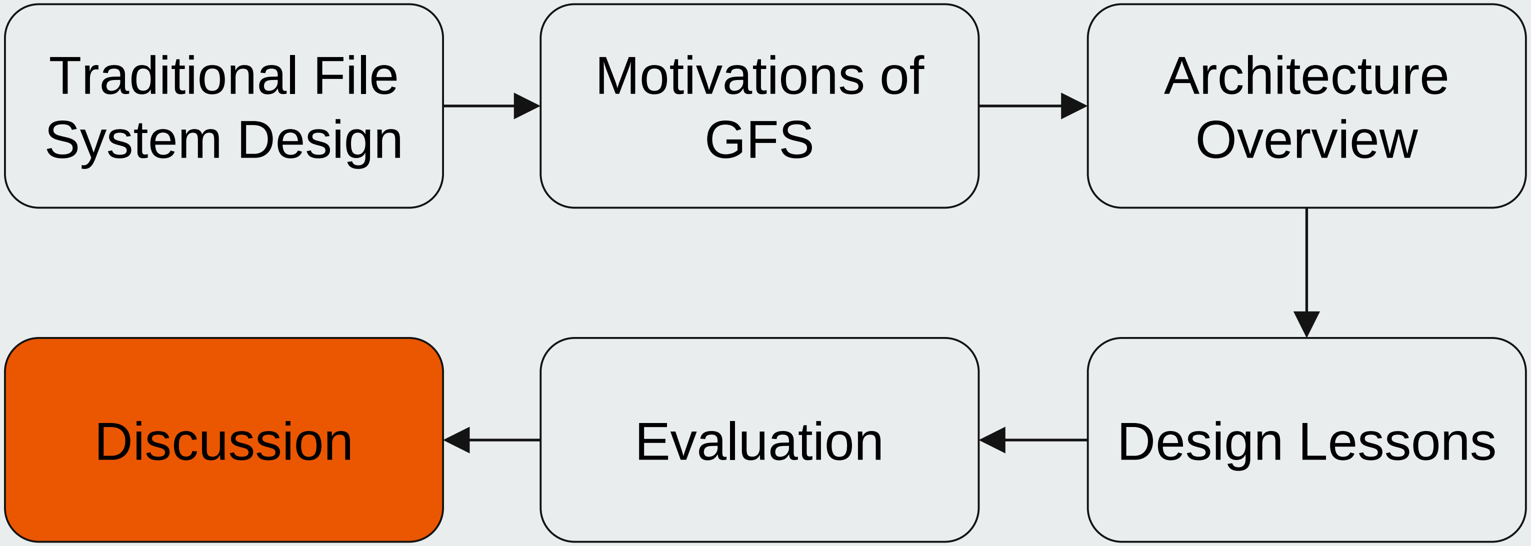
Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

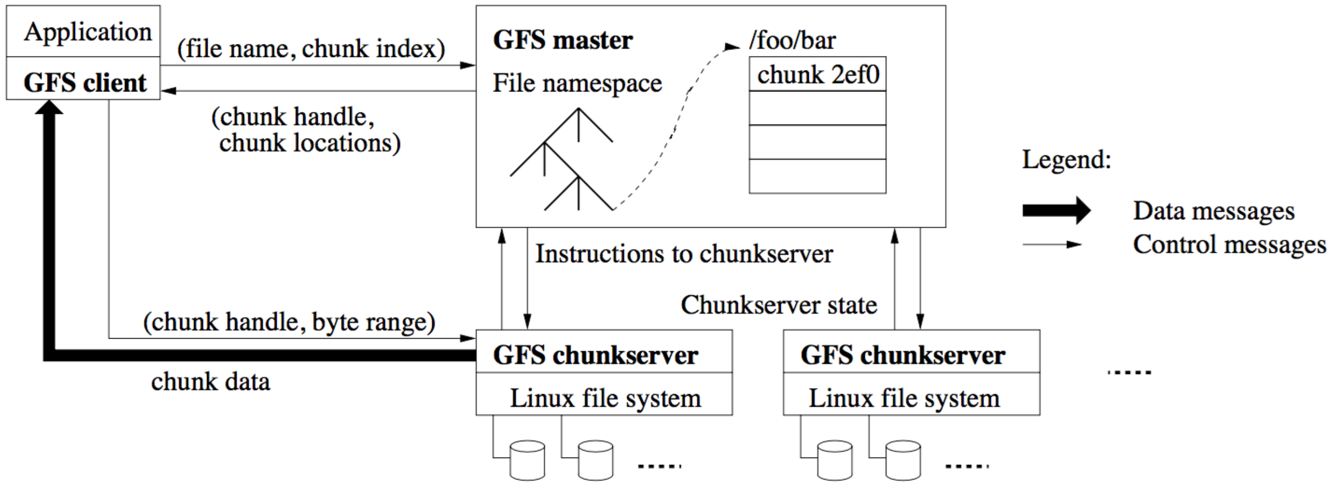
Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Roadmap


Traditional File System Design
Motivations of GFS
Architecture Overview
Design Lessons
Evaluation
Discussion

Discussion



Open Questions:

What if a chunk server is still overloaded?

Why using Linux file system? Recall Stonebraker's argument.

What are pros/cons of a single master in this system? How can single master be a problem?

Are industry papers useful to the rest of us? Details?

Spanner

Combining consistency and performance in a globally-distributed database

Authors

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

Background

- Bigtable
  - Another database designed by Google
  - Fast, but not strongly consistent
  - Limited support for transactions
- Megastore
  - Another database designed by Google
  - Strong consistency, but poor write throughput
- Can we get the best of both?

Cloud Spanner: The best of the relational and NoSQL worlds

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ **Yes**	✓ Yes	✗ No
SQL	✓ **Yes**	✓ Yes	✗ No
Consistency	✓ **Strong**	✓ Strong	✗ Eventual
Availability	✓ **High**	✗ Failover	✓ High
Scalability	✓ **Horizontal**	✗ Vertical	✓ Horizontal
Replication	✓ **Automatic**	Configurable	Configurable

What does Spanner do?

- Key-value store with SQL
- Transactions
- Globally distributed (why?)
- Externally consistent (why?)
- Fault-tolerant

Claim to fame

"It is the first system to distribute data at global scale and support externally-consistent distributed transactions."

What does Spanner do?

- **Key-value store with SQL**
  - Familiar database interface for clients
- **Transactions**
  - Perform several updates atomically
- **Globally distributed**
  - Can scale up to “millions of machines” across continents
  - Protection from wide-area disasters
- **Externally consistent**
  - Clients see a single sequential transaction ordering
  - This ordering reflects the order of the transactions in real time
- **Fault-tolerant**
  - Data is replicated across Paxos state machines

Why we want external consistency

- Transaction T_1 deposits \$200 into a bank account
- Transaction T_2 withdraws \$150
- If the bank observes a negative balance at any point, the customer incurs a penalty
- In this case, we want that no database read sees the effects of T_2 before it sees the effects of T_1

Example taken from the documentation for Cloud Spanner

TrueTime API

Method	Returns
$TT.now()$	$TTinterval$: $[earliest, latest]$
$TT.after(t)$	true if t has definitely passed
$TT.before(t)$	true if t has definitely not arrived

Basic idea

- Transactions are ordered by timestamps that correspond to real time
- In order to maintain consistency across replicas, a Spanner node artificially delays certain operations until it is sure that a particular time has passed on all nodes

TrueTime API

- Previously, distributed systems could not rely on synchronized clock guarantees
  - Sending time across the network is tricky
- Google gets around this by using atomic clocks (referred to as "Armageddon masters") and GPS clocks

"As a community, **we should no longer depend on loosely synchronized clocks** and weak time APIs in designing distributed algorithms."

TrueTime API

Key benefits

- Paxos leader leases can be made long-lived and disjoint
  - No contention--good for performance!
- External consistency can be enforced
  - Two-phase locking can also enforce external consistency, but even read-only transactions must acquire locks. On the other hand, Spanner maintains multiple versions of key-value mapping, and uses TrueTime to allow read-only transactions and snapshot reads to commit without locks. This makes performance practical.

Locality

- Data is sharded using key prefixes
- `(userID, albumID, photoID) -> photo.jpg`
- The data for a particular user is likely to be stored together

Evaluation

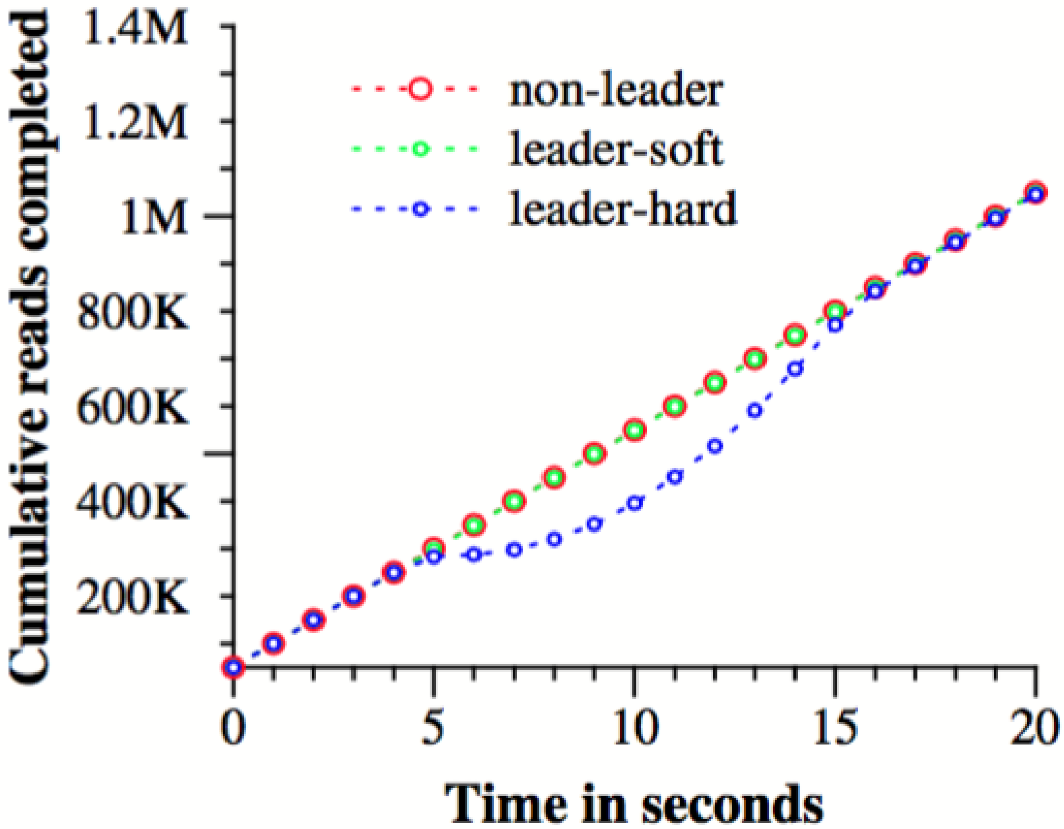
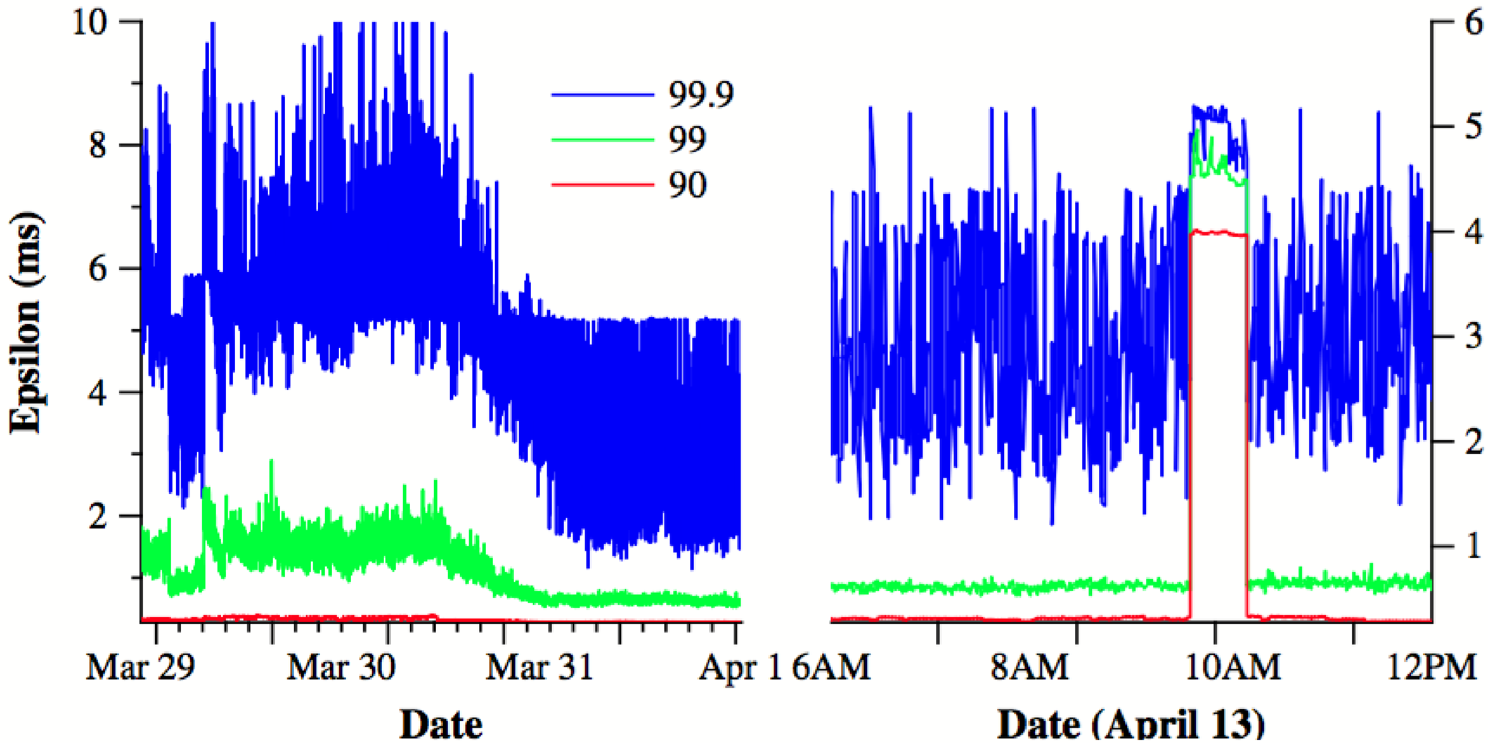


Figure 5: Effect of killing servers on throughput.

Evaluation



10
8
6
4
2
Epsilon (ms)
99.9
99
90
Mar 29
Mar 30
Mar 31
Apr 1
Date
6AM
8AM
10AM
12PM
Date (April 13)
6
5
4
3
2
1

Closing Remarks

- **Assumptions guide design**
    - E.g., GFS is optimized for large sequential reads
    - E.g., Spanner is built for applications that need strong consistency
- **Fast, consistent, global replication of data is possible**
    - Just need careful design (and maybe atomic clocks!)

Closing Remarks

“In a production environment we cannot overstate the strength of a design that is straight-forward to implement and to maintain”

-- *Finding a needle in Haystack: Facebook’s photo storage*