The Google Storage Stack (Chubby, GFS, BigTable)

Dan Ports, CSEP 552

Today

- Three real-world systems from Google
  - GFS: large-scale storage for bulk data
  - BigTable: scalable storage of structured data
  - Chubby: coordination to support other services

- Each of these systems has been quite influential

- Lots of open-source clones:
  GFS -> HDFS
  BigTable -> HBase, Cassandra, etc
  Chubby -> ZooKeeper

- Also 10+ years old
  (published 2003/2006; in use for years before that)

  - major changes in design & workloads since then

These are real systems

- Not necessarily the best design
- Discussion topics:
  - are these the best solutions for their problem?
  - are they even the right problem?
- Lots of interesting stories about side problems from real deployments

Chubby

- One of the first distributed *coordination services*
- Goal: allow client apps to synchronize themselves and manage info about their environment
  - e.g., select a GFS master
  - e.g., find the BigTable directory
  - e.g., be the view service from Lab 2
- Internally: Paxos-replicated state machine

Chubby History

- Google has a lot of services that need reliable coordination; originally doing ad-hoc things
- Paxos is a known-correct answer, but it's hard!
- build a service to make it available to apps
- actually: first attempt did not use Paxos
  - Berkeley DB replication — this did not go well

Chubby Interface

- like a simple file system
- hierarchical directory structure: /ls/cell/app/file
  - files are small: ~1KB
- Open a file, then:
  - GetContents, SetContents, Delete
  - locking: Acquire, TryAcquire, Release
  - sequencers: Get/Set/CheckSequencer

Example: Primary Election

```
x = Open(“/ls/cell/service/primary")
if (TryAcquire(x) == success) {
    // I'm the primary, tell everyone
    SetContents(x, my-address)
} else {

    // I'm not the primary, find out who is
    primary = GetContents(x)
    // also set up notifications
    //in case the primary changes
}
```

Why this interface?

- Why not, say, a Paxos consensus libray?

- Developers do not know how to use Paxos
  (they at least think they know how to use locks!)

- Backwards compatibility

- Want to advertise results outside of the system
  e.g., let all the clients know where the BigTable root is, not just the replicas of the master

- Want a separate set of nodes to run consensus
  like the view service in Chain Replication

State Machine Replication

- system state and output entirely determined by input

- then replication just means agreeing on order of inputs (and Paxos show us how to do this!)

- Limitations on system:
  - deterministic: handle clocks/randomness/etc specially
  - parallelism within a server is tricky
  - no communication except through state machine ops

- Great way to build a replicated service from scratch, really hard to retrofit to an existing system!

Implementation

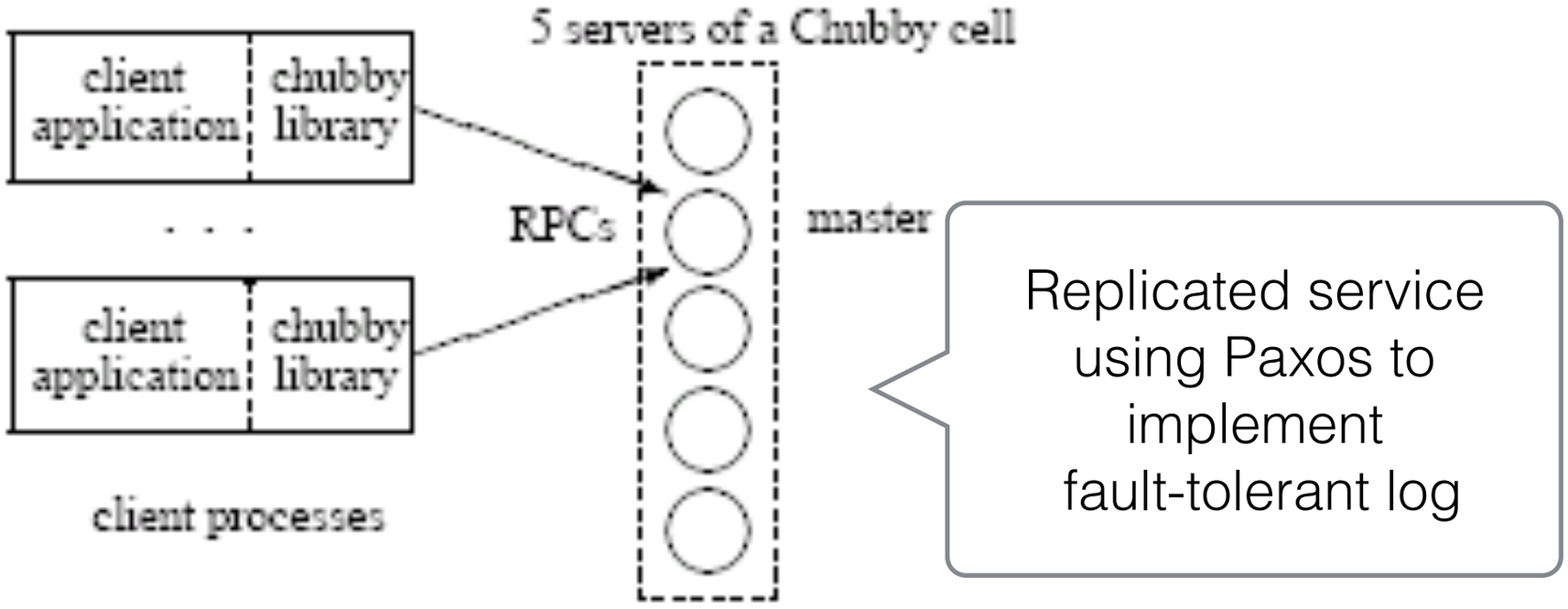
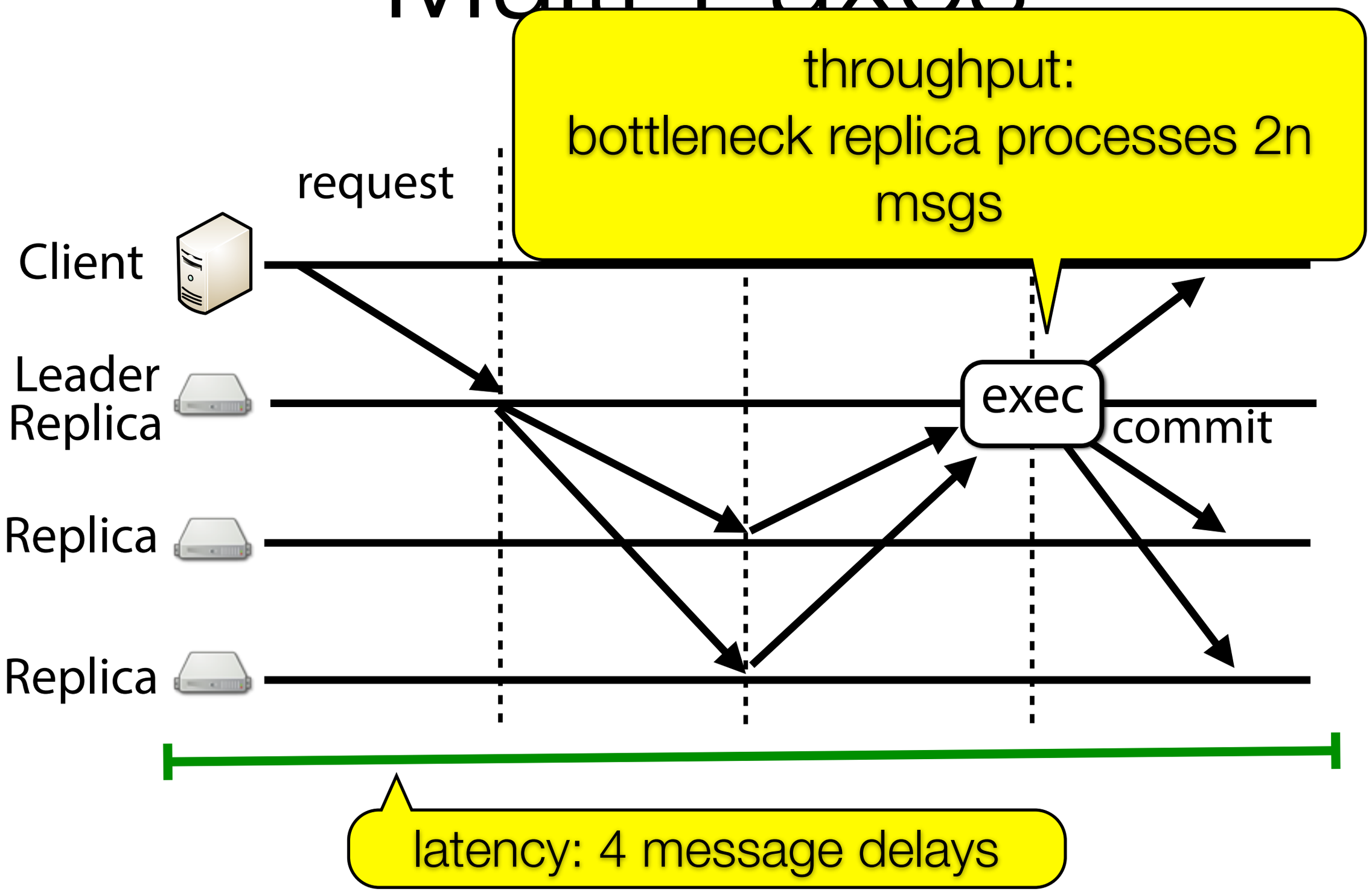


Figure 1: System structure

Challenge: performance!

- note: Chubby is not a high-performance system!
- but server does need to handle ~2000-5000 RPC/s
- Paxos implementation: < 1000 ops/sec
- …so can’t just use Paxos/SMR out of the box
- …need to engineer it so we don’t have to run Paxos on every RPC

Multi-Paxos


throughput:
bottleneck replica processes 2n msgs
request
Client
Leader
Replica
Replica
Replica
exec
commit
latency: 4 message delays

Paxos performance

- Last time: batching and partitioning
- Other ideas in the paper: leases, caching, proxies
- Other ideas?

Leases

- In a Paxos system (and in Lab 2!),
  the primary can't unilaterally respond to any request,
  including reads!

- Usual answer: use coordination (Paxos) on every request,
  including reads

- Common optimization: give the leader a lease
  for ~10 seconds, renewable

- Leader can process reads alone, if holding lease

- What do we have to do when the leader changes?

Caching

- What does Chubby cache?
  - file data, metadata — including absence of file
- What consistency level does Chubby provide?
  - strict consistency: linearizability
  - is this necessary? useful?
    (Note that ZooKeeper does not do this)

Caching implementation

- Client maintains local cache
- Master keeps a list of which clients might have each file cached
- Master sends invalidations on update (not the new version — why?)
- Cache entries have leases: expire automatically after a few seconds

Proxies

- Most of the master's load turns out to be keeping track of clients
  - keep-alive messages to make sure they haven't failed
  - invalidating cache entries
- Optimization: have groups of clients connect through a proxy
  - then the proxy is responsible for keeping track of which ones are alive and who to send invals to
  - can also adapt to different protocol format

Surprising use case

"Even though Chubby was designed as a lock service, we found that its most popular use was as a name server."

e.g., use Chubby instead of DNS to track hostnames for each participant in a MapReduce

DNS Caching vs Chubby

- DNS caching:
  purely time-based: entries expire after N seconds

- If too high (1 day): too slow to update;
  if too low (60 seconds): caching doesn't help!

- Chubby: clients keep data in cache, server invalidates them when it changes

  - much better for infrequently-updated items
    if we want fast updates!

- Could we replace DNS with Chubby everywhere?

Client Failure

- Clients have a persistent connection to Chubby
- Need to acknowledge it with periodic keep-alives (~10 seconds)

- If none received, Chubby declares client dead, closes its files, drops any locks it holds, stops tracking its cache entries, etc

Master Failure

- From client's perspective:
  - if haven't heard from the master,
    tell app session is in jeopardy;
    clear cache, client operations have to wait
  - if still no response in grace period (~45 sec),
    give up, assume Chubby has failed
    (what does the app have to do?)

Master Failure

- Run a Paxos round to elect a new master
- Increment a master epoch number (view number!)
- New master receives log of old operations committed by primary (from backups)
    - rebuild state: which clients have which files open, what's in each file, who holds which locks, etc
- Wait for old master's lease to expire
- Tell clients there was a failover (why?)

Performance

- ~50k clients per cell
- ~22k files — majority are open at a time
  most less than 1k; all less than 256k
- 2K RPCs/sec
  - but 93% are keep-alives, so caching, leases help!
  - most of the rest are reads, so master leases help
  - < 0.07% are modifications!

“Readers will be unsurprised to learn that the fail-over code, which is exercised far less often than other parts of the system, has been a rich source of interesting bugs.”

"In a few dozen cell-years of operation, we have lost data on six occasions, due to database software errors (4) and operator error (2); none involved hardware error."

“A related problem is the lack of performance advice in most software documentation. A module written by one team may be reused a year later by another team with disastrous results. It is sometimes hard to explain to interface designers that they must change their interfaces not because they are bad, but because other developers may be less aware of the cost of an RPC.”

GFS

- Google needed a distributed file system for storing search index
  (late 90s, paper 2003)

- Why not use an off-the-shelf FS? (NFS, AFS, …

  - very different workload characteristics!

  - able to design GFS for Google apps
    and design Google apps around GFS

GFS Workload

- Hundreds of web crawling clients
- Periodic batch analytic jobs like MapReduce
- Big data sets (for the time):
  1000 servers, 300 TB of data stored
- Note that this workload has changed over time!

GFS Workload

- few million 100MB+ files
  nothing smaller; some huge
- reads: small random and large streaming reads
- writes:
  - many files written once; other files appended to
  - random writes not supported!

GFS Interface

- app-level library, not a POSIX file system
- create, delete, open, close, read, write
    - concurrent writes not guaranteed to be consistent!
- record append: guaranteed to be atomic
- snapshots

Life without random writes

- E.g., results of a previous crawl:

  www.page1.com -> www.my.blogspot.com

  www.page2.com -> www.my.blogspot.com

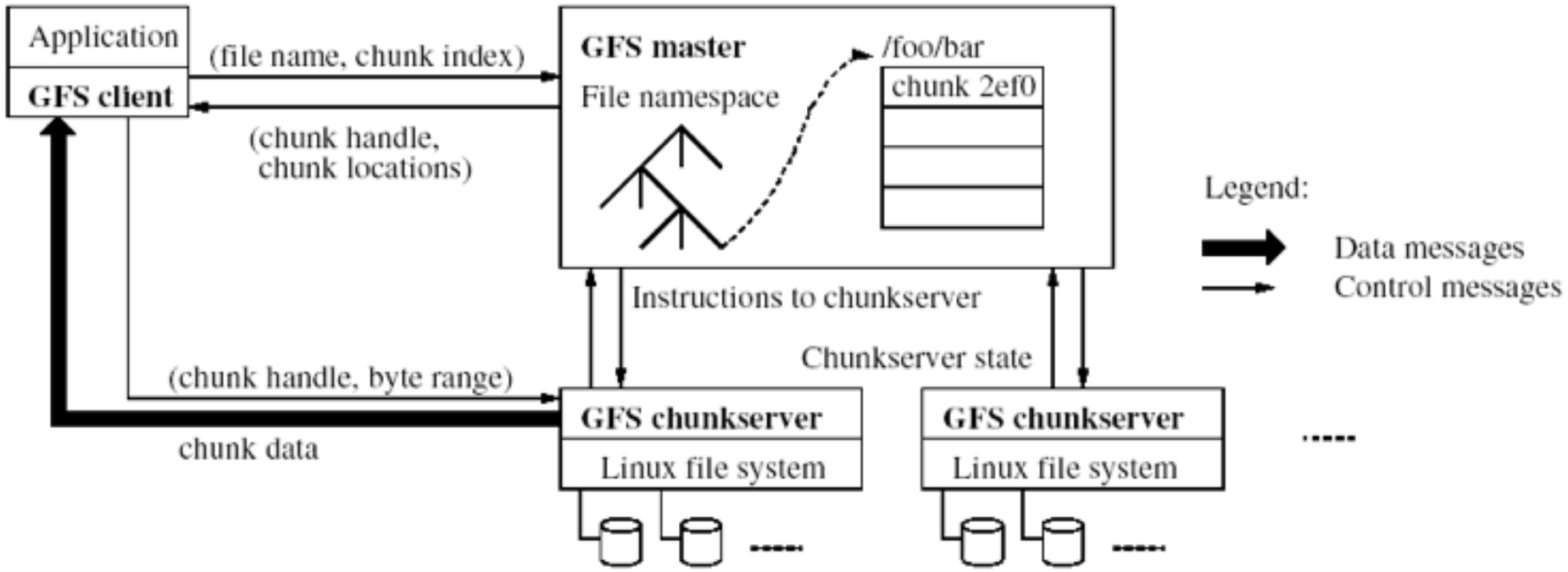
- Let's say new results: page2 no longer has the link, but there is a new page, page3:

  www.page1.com -> www.my.blogspot.com

  www.page3.com -> www.my.blogspot.com

- Option: delete the old record (page2), and insert a new record (page3)
  - requires locking, hard to implement
  - GFS way: delete the old file,
    create a new file where program can append new records to the file atomically

GFS Architecture



- each file stored as 64MB chunks
- each chunk on 3+ chunkservers
- single master stores metadata

“Single” Master Architecture

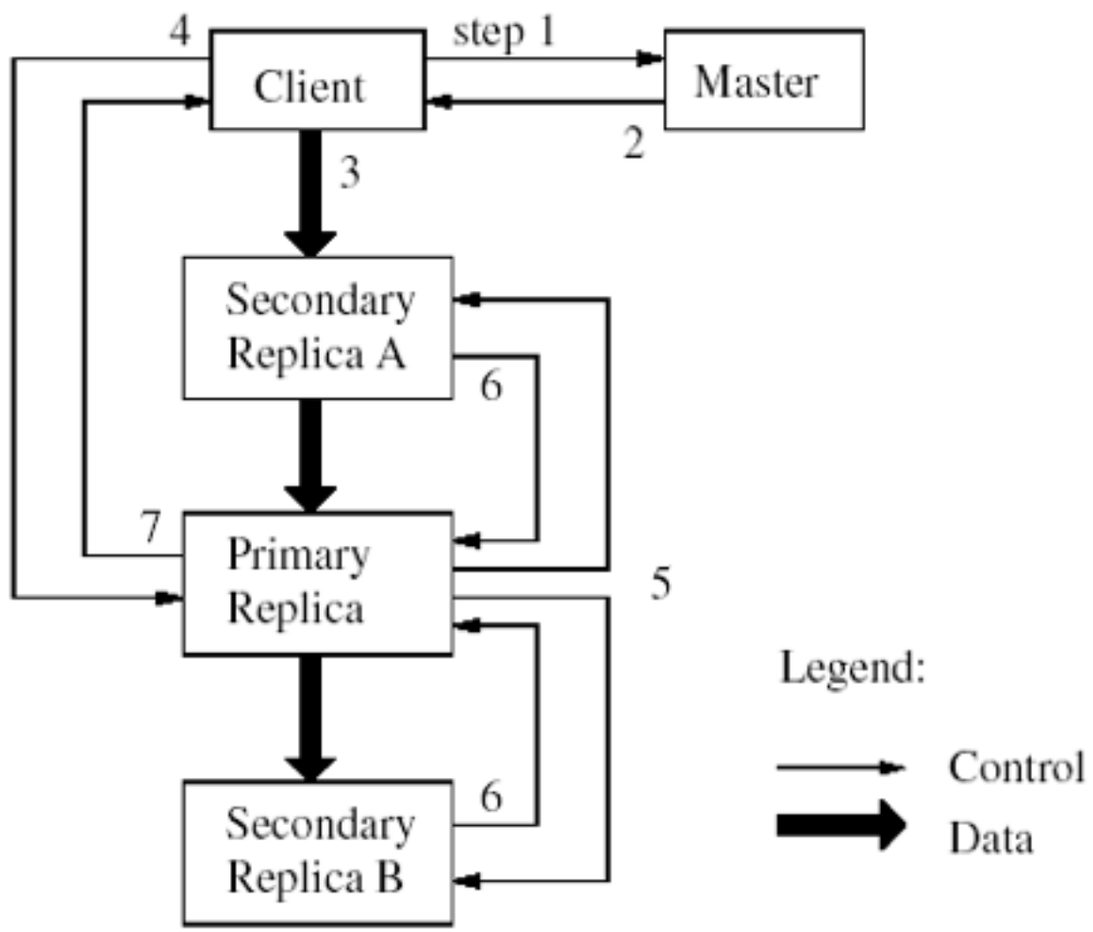
- Master stores metadata:
  file name -> chunk list
  chunk ID -> list of chunkservers holding it
- All metadata stored in memory (~64B/chunk)
- Never stores file *contents!*
- Actually a replicated system using shadow masters

Master Fault Tolerance

- One master plus set of replicas
- Master chosen w/ Chubby
- Master keeps a log of operations that update metadata
- Each log write has to be acknowledged by shadow masters before executing operation
- Periodic checkpoints of log state
  i.e., take a snapshot of DB then switch to a new log

Handling Write Operations

- Mutation is write or append
- Goal: minimize master involvement
- Lease mechanism
  - Master picks one replica as primary; gives it a lease
  - Primary defines a serial order of mutations
- Data flow decoupled from control flow



Write Operations

- Application originates write request
- GFS client translates request from (fname, data) --> (fname, chunk-index) sends it to master
- Master responds with chunk handle and (primary +secondary) replica locations
- Client pushes write data to all locations; data is stored in chunkservers' internal buffers
- Client sends write command to primary

Write Operations (contd.)

- Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk

- Primary sends serial order to the secondaries and tells them to perform the write

- Secondaries respond to the primary

- Primary responds back to client

- Note: if write fails at one of the chunkservers, client is informed and retries the write

Caching

- GFS does not use caching on the clients!
- Chubby said that caching was essential
- What's different here?

Discussion

- Is this a good design?
- Can we improve on it?
- Will it scale to even larger workloads?

~15 years later

- Scale is much bigger:
  now 10K servers instead of 1K
  now 100 PB instead of 100 TB

- Bigger workload change:
  not everything is batch updates to small files!

  - ~2010: Google moves to incremental updates of the index instead of periodically rebuilding it w/ MapReduce

GFS Endgame

- GFS scaled to ~50 million files, ~10 PB
- Developers had to organize their apps around large append-only files (see BigTable)
- Latency-sensitive applications suffered
- GFS eventually replaced with a new design, Colossus

Discussion

- What would you change to fix these problems?

Metadata scalability

- Main scalability limit: single master stores all metadata
- HDFS has same problem (single NameNode)
- Approach: partition the metadata among multiple masters
- New system supports ~100M files per master and smaller chunk sizes: 1MB instead of 64MB

Reducing Storage Overhead

- Replication: 3x storage to handle two copies
- Erasure coding more flexible: m pieces, n check pieces
  - e.g., RAID-5: 2 disks, 1 parity disk (XOR of other two) => 1 failure w/ only 1.5 storage
- Sub-chunk writes more expensive (read-modify-write)
- Recovery is harder:
  usually need to get all the other pieces,
  generate another one after the failure

Erasure Coding

- 3-way replication:
  3x overhead, 2 failures tolerated, easy recovery

- Google Colossus: (6,3) Reed-Solomon code
  1.5x overhead, 3 failures

- Facebook HDFS: (10,4) Reed-Solomon
  1.4x overhead, 4 failures, expensive recovery

- Azure: more advanced code (12, 4)
  1.33x, 4 failures, same recovery cost as Colossus

BigTable

- stores (semi)-structured data
  - e.g., URL -> contents, metadata, links
  - e.g., user > preferences, recent queries

- really large scale!
  - capacity: 100 billion pages * 10 versions => 20PB
  - throughput: 100M users, millions of queries/sec
  - latency: can only afford a few milliseconds per lookup

Why not use a commercial DB?

- Scale is too large, and/or cost too high
- Low-level storage optimizations help
  - data model exposes locality, performance tradeoff
  - traditional DBs try to hide this!
- Can remove “unnecessary” features
  - secondary indexes, multirow transactions, integrity constraints

Data Model

- a big, sparse, multidimensional sorted table
- (row, column, timestamp) -> contents
- fast lookup on a key
- rows are ordered lexicographically, so scans in order

Consistency

- Is this an ACID system?
- Durability and atomicity: via commit log in GFS
- Strong consistency:
  operations get processed by a single server in order
- Isolated transactions:
  single-row only, e.g., compare-and-swap

Implementation

- Divide the table into tablets (~100 MB) grouped by a range of sorted rows

- Each tablet is stored on a tablet server that manages 10-1000 tablets

- Master assigns tablets to servers, reassigns when servers are new/crashed/overloaded, splits tablets as necessary

- Client library responsible for locating the data

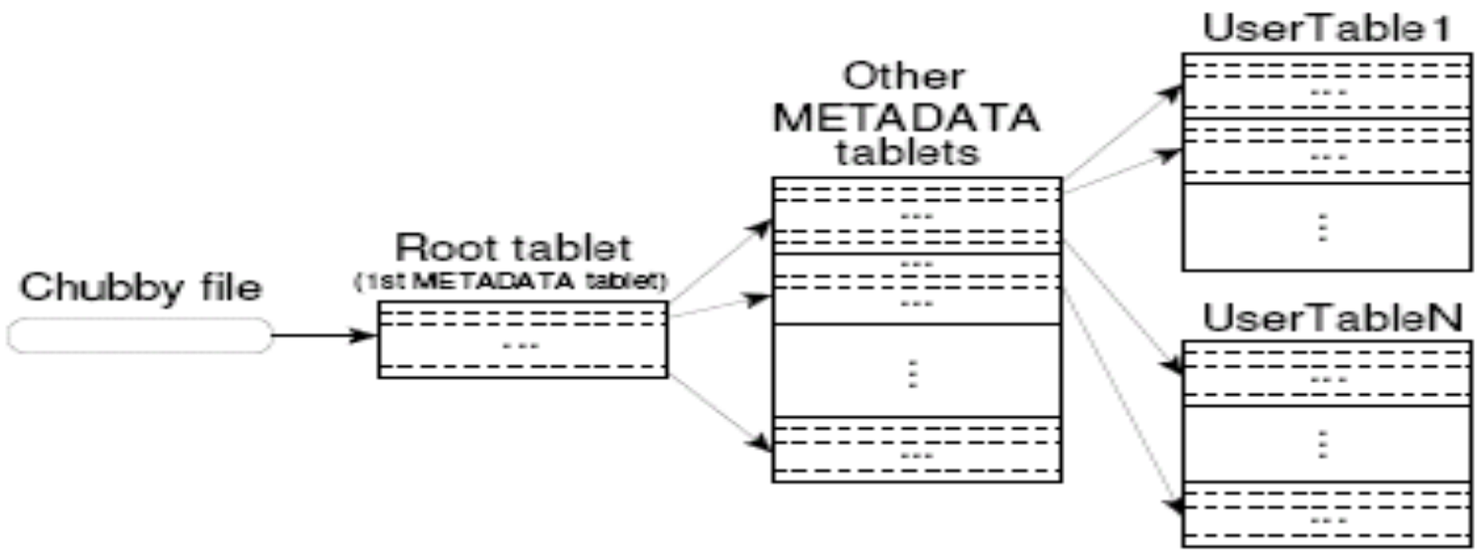
Is this just like GFS?

Is this just like GFS?

- Same general architecture, but…
- can leverage GFS and Chubby!
- tablet servers and master are basically stateless
  - tablet data is stored in GFS, coordinated via Chubby
  - master serves most config data in Chubby

Is this just like GFS?

- Scalable metadata assignment
- Don’t store the entire list of row -> tablet -> server mappings in the master
- 3-level hierarchy
  entries are location: ip/port of relevant server



Storage on a tablet

- Most data stored in set of “SSTables”:
  sorted key-value pairs on disk

- new writes are added to in-memory “memtable”
  + a log in GFS

- periodically move data form the memtable to disk

  - basically: read SSTables in, remove old values, merge in new data from memtable, write back out

  - see LSM-trees, LevelDB

Fault tolerance

- If a tablet server fails (while storing ~100 tablets)
  - reassign each tablet to another machine
  - so 100 machines pick up just 1 tablet each
    - tablet SSTables & log are in GFS

- If the master fails
  - acquire lock from Chubby to elect new master
  - read config data from Chubby
  - contact all tablet servers to ask what they're responsible for

BigTable in retrospect

- Definitely a useful, scalable system!
- Still in use at Google, motivated lots of NoSQL DBs
- Biggest mistake in design (per Jeff Dean, Google): not supporting distributed transactions!
  - became really important w/ incremental updates
  - users wanted them, implemented themselves, often incorrectly!
  - at least 3 papers later fixed this — two next week!