Chubby and BigTable

Tom Anderson
(based on slides from Dan Ports)

Outline

Last time:
- Memcache: Facebook caching layer

Today/Friday/Monday:
- Chubby: coordination service
- BigTable: scalable storage of structured data
- GFS: large-scale storage for bulk data

Impact

- Chubby/BigTable/GFS were the basis of Google's storage stack
  - Each 10+ years old
  - major changes in design & workloads since then
- Inspired related projects at Microsoft, Amazon, …
- Open-source versions:
  - GFS -> HDFS
  - BigTable -> HBase, Cassandra, etc
  - Chubby -> ZooKeeper

Chubby

- One of the first distributed *coordination services*
- Goal: allow client apps to synchronize and manage dynamic configuration state
  - A highly available view service for Lab 2
  - Find the BigTable directory
  - Select a GFS master
- Internally: Paxos-replicated state machine

Chubby History

- Replace ad hoc solutions to coordination
- Similar problem at many companies
  - Chubby at Google
  - Distributed lock service at Amazon
  - Zookeeper for open source
- Paxos is well-known correct answer
  - Chubby provides Paxos as a service to apps

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
- Want to advertise results outside of the system e.g., let all clients know where to find BigTable root, not just the replicas of the master
- Consensus as a service, not a library

Implementation

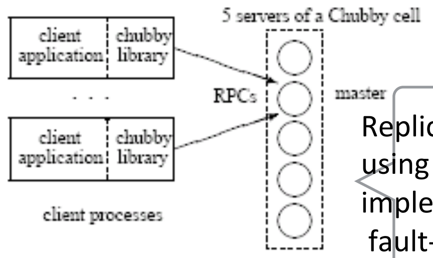


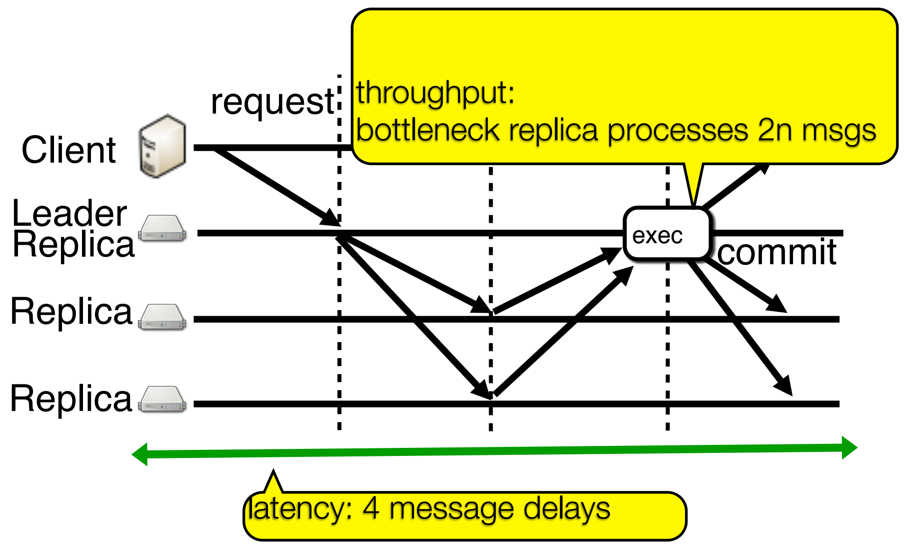
Figure 1: System structure

Replicated service using Paxos to implement fault-tolerant log

What About Performance?

- Chubby is not a high-performance system!
- Paxos implementation: < 1000 ops/sec
- Initial version: needed to handle ~2000-5000 RPC/s
- Scale by adding nodes to Paxos group?

Multi-Paxos


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

Paxos Performance Optimizations

- Batching
- Partitioning
- Leases
- Caching
- Proxies
- Other ideas?

Batching and Partitioning

- Batching
  - Have leader accumulate requests from many clients
  - Run one round of Paxos to add them all to the log
  - Much higher throughput, somewhat higher latency
- Partitioning
  - Run multiple Paxos groups to spread load
  - Each replica will be a leader in some, follower in others
  - Very common in practice

Leases

- In Paxos (and lab 2), the primary can't unilaterally respond to requests, 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 happens when the leader changes?

Caching

- What does Chubby cache?
  - file data, metadata — including absence of file
- 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: connect groups of clients through a proxy
  - Proxy keeps track of which ones are alive and who needs invalidations

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 vs. Chubby

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

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

- Chubby: server invalidates clients when needed
  - much better for infrequently-updated items
- 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

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."

"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."

BigTable

- Key-value store for (semi)-structured versioned data
  - e.g., URL -> contents, metadata, links
  - e.g., user > preferences, recent queries
- Very large scale!
  - capacity: 100 billion pages * 10 versions => 20PB
  - throughput: 100M users, millions of queries/sec
  - latency: a few milliseconds per lookup

Why Not Use a Commercial Database?

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

Key Ideas

Unstructured key-value table data
- No need for having a schema in advance
- instead create columns when needed

Versioned data, with key-specific garbage collection

Cluster data used together on same tablet

Instead of consistent hashing, reconfigure tablet boundaries for load balancing

Tablets for indexing into key space

Efficient updates using log structure (store deltas)

Data Model: Key-value++

- a big, sparse, multidimensional sorted table
- (row, column, timestamp) -> contents
- All data can be versioned, and GC'ed per key
- rows are ordered lexicographically, so
  - www.cs.washington.edu -> edu.washington.cs.www
  - Scans occur in order
  - Related data found nearby

Is BigTable ACID?

- Durability and atomicity: via GFS
- Strong consistency: operations processed by a single server in order
- Isolated transactions within a single key
- Multi-key transactions added in Spanner

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