Today’s Papers


- Thoughts?

---

## Google Chubby

- **A coarse-grained lock and small file storage service**
  - Other (Google) distributed systems can use this to synchronize access to shared resources

- Intended for use by “loosely-coupled distributed systems”
  - GFS: Elect a master
  - Bigtable: master election, client discovery, table service locking
  - Well-known location for bootstrapping larger systems
  - Partitioning workloads

- **Goals:**
  - High availability
  - Reliability

- **Anti-goals:**
  - High performance, Throughput, Storage capacity

---

## Distributed Consensus

![Diagram of Distributed Consensus](image)

- Chubby cell is usually 5 replicas
  - 3 must be alive for cell to be viable

- How do replicas in Chubby agree on their own master, official lock values?
  - Distributed commit algorithm
What about Two Phase Commit?

- Commit request/Voting phase
  - Coordinator sends query to commit
  - Cohorts prepare and reply – single abort vote causes complete abort

- Commit/Completion phase
  - Success: Commit and acknowledge
  - Failure: Rollback and acknowledge

- Disadvantage: Blocking protocol
  - Handles coordinator failures really poorly – blocks
  - Handles cohort failure poorly during voting phase – aborts

Basic Paxos (Quorum-based Consensus)

- Prepare and Promise
  - Proposer selects proposal number $N$ and sends promise to acceptors
  - Acceptors accept or deny the promise

- Accept! and Accepted
  - Proposer sends out value
  - Acceptors respond to proposer and learners

- Paxos algorithm properties
  - Family of algorithms (by Leslie Lamport) designed to provide distributed consensus in a network of several replicas
  - Enables reaching consensus on a single binding of variable to value ("fact")
  - Tolerates delayed or reordered messages and replicas that fail by stopping
  - Tolerates up to $N/2$ replica failure (i.e., $F$ faults with $2F+1$ replicas)

Message Flow: Basic Paxos

- Proposer – An agent that proposes a fact
- Leader – the authoritative proposer
- Acceptor – holds agreed-upon facts in its memory
- Learner – May retrieve a fact from the system

```
Client  Proposer  Acceptor  Learner
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X------</td>
<td>--------</td>
<td>---------</td>
<td>---</td>
</tr>
<tr>
<td>Request</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
Propose(N, {V_a, V_b, V_c})
```

```
| X------|--------|---------|---|
| Accept!|         |         |   |

Paxos Assumptions

- Replica assumptions
  - Operate at arbitrary speed
  - Independent, random failures
  - Replicas with stable storage may rejoin protocol after failure
  - Do not lie, collude, or attempt to maliciously subvert the protocol

- Network assumptions
  - All processors can communicate ("see") one another
  - Messages are sent asynchronously and may take arbitrarily long to deliver
  - Order of messages is not guaranteed: they may be lost, reordered, or duplicated
  - Messages, if delivered, are not corrupted in the process
Basic Paxos – Majority consensus

- Determines the authoritative value for a single variable
- Each proposer makes a proposal to some majority (quorum) of the acceptors; acceptors respond with latest value
- A majority (quorum) of acceptors must accept a proposal for the proposed value to be chosen as the consensus value
- If $P_1$ and $P_2$ are making different proposals, then there must be at least one acceptor that they share in common – this common acceptor decides which proposal prevails

Choosing a value

- Since different proposers (leaders) may be proposing at the same time, protocol uses disjoint proposal numbers (e.g., put identity in LSB)
- An acceptor will accept proposal with largest proposal number
- A value is chosen once majority (quorum) of acceptors have accepted a proposal with that value
- During failed rounds (not majority acceptance), responding acceptors keep track of their previous “votes” to help protocol converge on single value – even in presence of multiple proposers (leaders)

Step 1: Prepare

- Proposer 1
  - PREPARE $j$
- Proposer 2
  - PREPARE $k$
- Acceptors
  - $k > j$

Step 2: Promise

- PROMISE $x$ – Acceptor will accept proposals only numbered $x$ or higher
- Proposer 1 is ineligible because a quorum has voted for a higher number than $j$
MultiPaxos

- Within each instance (basic) Praxos is used to arrive at a consensus of the value to be used by all replicas
- The sequence of instances determines a sequence of values accepted by all replicas

Paxos in Chubby

- MultiPaxos:
  - Steps 1 (prepare) and 2 (promise) done once
  - Steps 3 (accept!) and 4 (accepted) repeated multiple times by same leader

- Replicas in a cell initially use Paxos to establish the leader
  - Majority of replicas must agree

- Optimization: Master Lease
  - Replicas promise not to try to elect new master for at least a few seconds
  - Master lease is periodically renewed

- Master failure
  - If replicas lose contact with master, they wait for grace period (4-6 secs)
  - On timeout, hold new election
From Theory to Practice: Fault-tolerant LOG implement with Paxos

- **Disk Corruption**
  - Need to recognize and handle subtle corruption in stable state

- **Use of Master Leases**
  - Grant leadership for fixed period of time
  - Allows clients to read latest value from Master
  - Prevents inefficient oscillation in algorithm

- **Use of Epochs**
  - Recognize when leader changes
  - Chubby semantics requires abort in these circumstances

- **Group membership**
  - Use of Paxos protocol to select servers that are members of Paxos group

- **Snapshot integration with Paxos**

- **MultiOp**
  - Allows implementation of atomic operations on log
  - If (guard(database)) then {t_op} else {f_op}

Building a Correct System

- Simple one-page pseudocode for Paxos algorithm == thousands of lines of C++ code
  - Created simple state machine specification language and compiler
  - Resulting code is “Correct by construction”

- Aggressive testing strategy
  - Tests for safety (consistent) and liveness (consistent and making progress)
  - Added entry points for test harnesses
  - Reproducible simulation environment
    - Injection of random errors in network and hardware
    - Use of pseudo-random seed provided reproducibility

- Data structure and database corruption
  - Aggressive, liberal usage of `assert` statements (makes Chubby fail-stop)
  - Added lots of checksum checks

- Upgrades and rollbacks are hard
  - Fix buggy scripts!
  - Recognize differences between developers and operators

- Forced to “add concurrency” as project progressed

Reliability

- Started out using replicated Berkeley DB (“3DB”)
  - Ill-defined, unproven, buggy replication protocol

- Replaced with custom write-thru logging DB

- Entire database periodically sent to GFS
  - In a different data center

- Chubby replicas span multiple racks
Summary

• Simple protocols win again

• Reuse of functionality
  – Chubby uses GFS
  – GFS uses Chubby

• Many challenges going from theoretical algorithm to practical implementation
  – No tools for implementing fault-tolerant protocols
  – Test, test, and test again (critical component!)
  – Everything can be corrupted so checksum everything
  – People are fallible (so are scripts!)

Is this a good paper?

• What were the authors’ goals?
• What about the evaluation/metrics?
• Did they convince you that this was a good system/approach?
• Were there any red-flags?
• What mistakes did they make?
• Does the system/approach meet the “Test of Time” challenge?
• How would you review this paper today?

Google Megastore – Motivation

• Storage requirements of today’s interactive online applications
  – Scalability (a billion internet users)
  – Rapid Development
  – Responsiveness (low latency)
  – Durability and Consistency (never lose data)
  – Fault Tolerant (no unplanned/planned downtime)
  – Easy Operations (minimize confusion, support is expensive)

• These requirements are in conflict!
### Technology Options

- **Scalable**
  - Highly scalable
  - Limited API
  - Loose consistency models
- **Eventual Consistency**
  - Rich feature set
  - Expressive language
  - Hard to scale
- **Wide-Area Replication**
- **ACID Transactions**
  - MySQL
  - Failover
  - Clustering
  - Quorum Voting
- **Bigtable**
- **Megastore**
  - Started in 2006 for app development at Google
  - Service layered on:
    - Bigtable (NoSQL scalable data store per datacenter)
    - Chubby (Config data, config locks)
  - Turnkey scaling (apps, users)
  - Developer-friendly features
  - Wide-area synchronous replication
    - Partition by "Entity Group"

### Megastore

- Service layered on:
  - Bigtable (NoSQL scalable data store per datacenter)
  - Chubby (Config data, config locks)
- Turnkey scaling (apps, users)
- Developer-friendly features
- Wide-area synchronous replication
  - Partition by "Entity Group"

### Data Model

- Between abstract tuples of RDBMS and concrete row-column storage of NoSQL
- Tables are *entity group root tables or child tables*
- Entity Group – consists of a *root entity* along with all *child entities*
- There can be several root tables – leading to several classes of *Entity Groups*

### Entity Groups

- Entity groups are sub-databases
**Entity Groups**

- Cheap transactions within an entity group (common)
  
  - Entity Group acts as mini-DB (ACID semantics)
  - Uses Write Ahead Logging
  - Store multiple data per cell by timestamp (Bigtable)
  - Multiversion concurrency using timestamps (Bigtable)

- Expensive or loosely-consistent operations across Entity Groups (rare)

**Entity Group Examples**

<table>
<thead>
<tr>
<th>Application</th>
<th>Entity Groups</th>
<th>Cross-EG Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email</td>
<td>User accounts</td>
<td>none</td>
</tr>
<tr>
<td>Blogs</td>
<td>Users, Blogs</td>
<td>Access control, notifications, global indexes</td>
</tr>
<tr>
<td>Mapping</td>
<td>Local patches</td>
<td>Patch-spanning ops</td>
</tr>
<tr>
<td>Social</td>
<td>Users, Groups</td>
<td>Messages, relationships, notifications</td>
</tr>
<tr>
<td>Resources</td>
<td>Sites</td>
<td>Shipments</td>
</tr>
</tbody>
</table>

**Achieving Technical Goals (I)**

- **Scalability**
  - Bigtable within datacenters – easy to add EGs (storage, throughput)
  - Performance maximized by partitioning based on EGs
  - Transactions within an EG – single phase using Paxos
  - Transactions across entity groups – two phase using Asynchronous Message Queue
  - Indexes – ACID within Entity Group, Looser semantics across EGs

- **Availability**
  - Fault Tolerance through replication
  - Fault Tolerant log replication of logs (adapted from Paxos)
Achieving Technical Goals (II)

- ACID transactions
  - Write-ahead log per Entity Group
  - 2PC or Queues between Entity Groups
- Wide-Area replication
  - Paxos with tweaks for optimal latency

Megastore’s Tweaks

- Coordinators
  - Tracks set of entity groups for which its replica has observed all Paxos writes
- Fast Reads
  - Local reads from any replica avoid inter-replica RPCs
  - Yield better utilization, low latencies in all regions, fine-grained read failover, simpler programming experience
- Fast Writes
  - Uses same pre-preparing optimization as Master approaches (accepted implies next prepare)
  - Uses leaders (coordinators) instead of masters and runs a Paxos instance for each log position – leader arbitrates which writer succeeds
- Replica Types
  - Witness Replicas: participate in voting (tie-breakers) and store log entries (no data)
  - Read-only Replicas: non-voting replicas containing snapshots

Megastore Architecture

Paxos and Megastore

- Basic Paxos not used (poor match for high-latency links)
  - Writes require at least two inter-replica roundtrips to achieve consensus (prepare round, accept round)
  - Reads require one inter-replica roundtrip (prepare round)
- Approaches using a Master replica
  - Master participates in all writes (state is always up-to-date)
  - Master serves reads (current consensus state) without additional comm
  - Writes are single roundtrip – piggyback prepare for next write on accepted
  - Batch writes for efficiency
- Issues with using a Master
  - Need to place transactions (readers) near master replica to avoid latency
  - Master must have sufficient processing resources (side effect: replicas waste resources since they must be capable of becoming masters)
  - Master failover requires lots of timers and a complex state machine (side effect: user visible outages)
Megastore Reads

Megastore Writes

Availability and Performance

Benefits

- For admins
  - Linear scaling, transparent rebalancing (Bigtable)
  - Instant transparent failover
  - Symmetric deployment

- For developers
  - ACID transactions (read-modify-write)
  - Many features (indexes, backup, encryption, scaling)
  - Single-system image makes code simple
  - Little need to handle failures

- For end Users
  - Fast up-to-date reads, acceptable write latency
  - Consistency
**Summary**

- Constraints acceptable for most apps
  - Entity Group partitioning
  - High write latency
  - Limited per-EG throughput

- In production use for over 4 years

- No current query language
  - Apps must implement query plans
  - Apps have fine-grained control of physical placement

- Available on Google App Engine as HRD (High Replication Datastore)

**Is this a good paper?**

- What were the authors’ goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the “Test of Time” challenge?
- How would you review this paper today?