Paxos Made Live

Tushar Chandra
Robert Griesemer
Joshua Redstone

Presented by

Jacob Kobernik

1. Intro

"We describe our experience building a fault-tolerant data-base using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system."

Building ... Paxos turned out to be a non-trivial task:

- "Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations"
- Proving correctness in a large-scale implementation was uncharted territory
- Algorithms tolerate a limited number of specific faults; the real world exposed the application to more which had to be handled
- "A real system is rarely specified precisely", getting from the napkin to production required tedious flexibility

- Table of Contents -

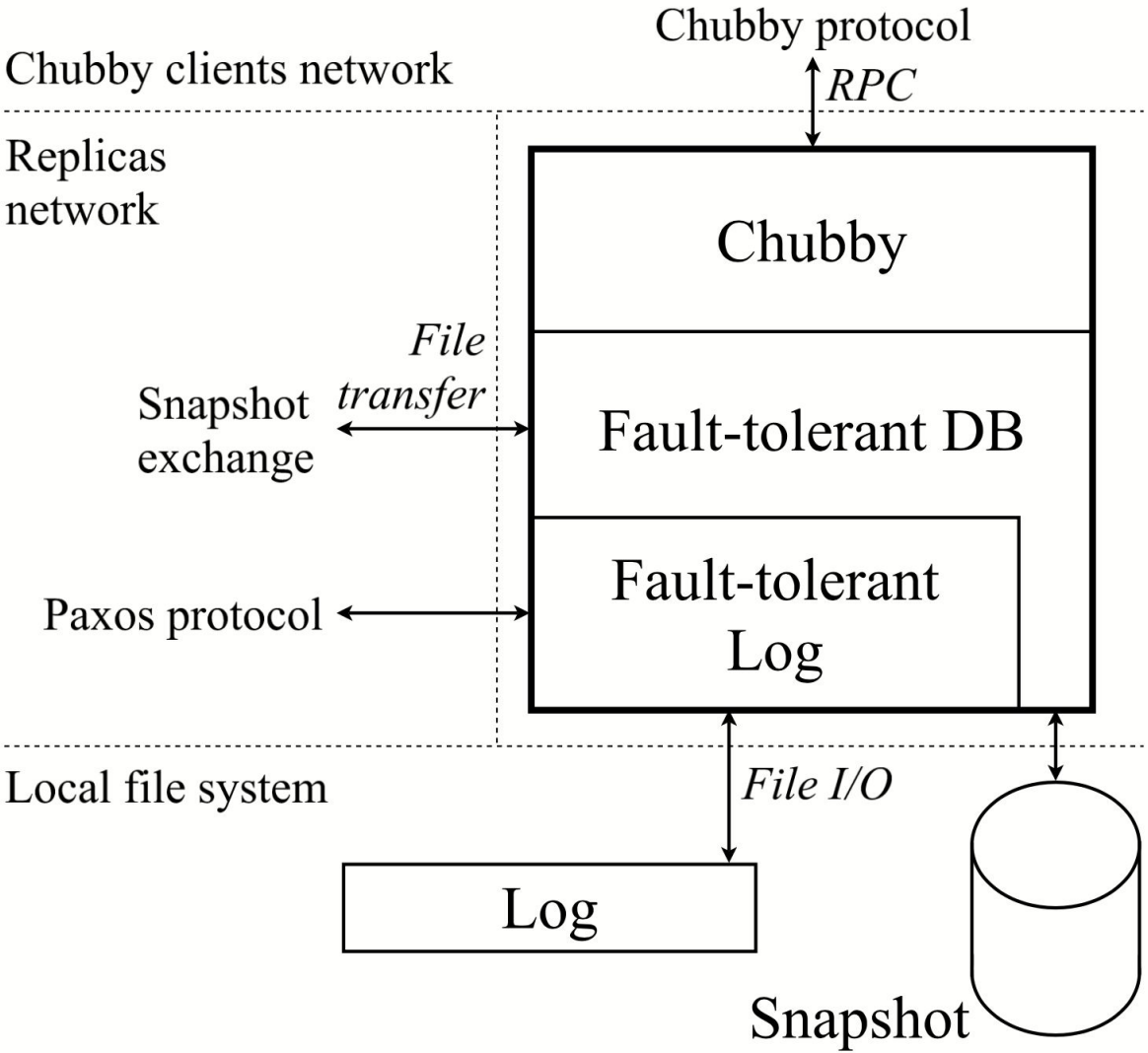
2. Chubby

Background

- Distributed locking mechanism at google
- Fault-tolerance is achieved through replication
- Often 1 chubby system at each data center
- 5 replicas comprise the system, called a "cell"
- Each replica runs the same code on a dedicated machine
- The cell is ruled by 1 replica playing the master role
- Any replica in a cell can take on the role of master at any time
- Usual clients are Google Filesystem and Big Table (distributed coordination)

Architecture

- Layered architecture for clarity and reusability
- Each layer has its own protocol system
- All built on top of the fault-tolerant log running Paxos


Chubby protocol
RPC
Chubby clients network
Replicas network
Chubby
File transfer
Snapshot exchange
Fault-tolerant DB
Fault-tolerant Log
Paxos protocol
Local file system
File I/O
Log
Snapshot

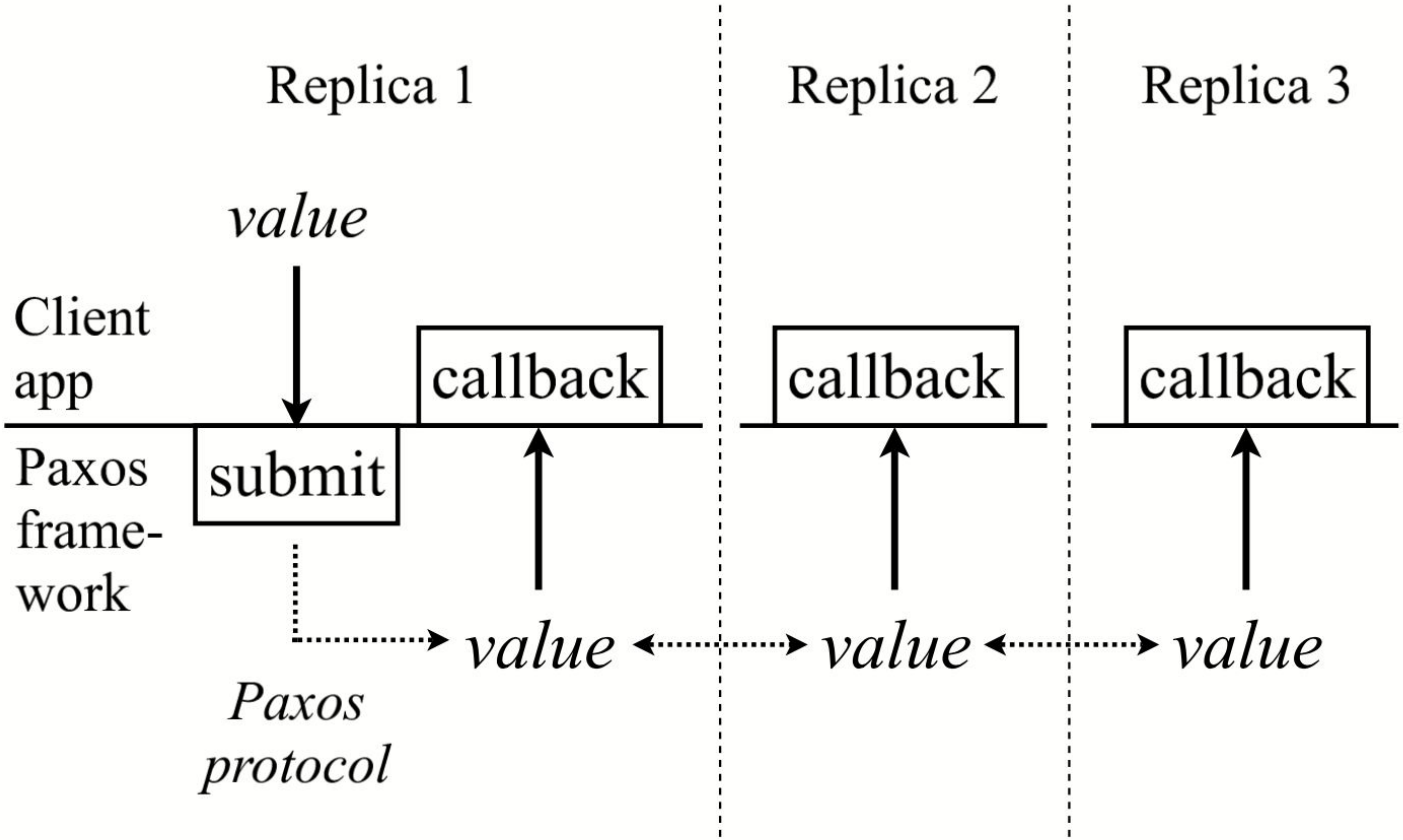
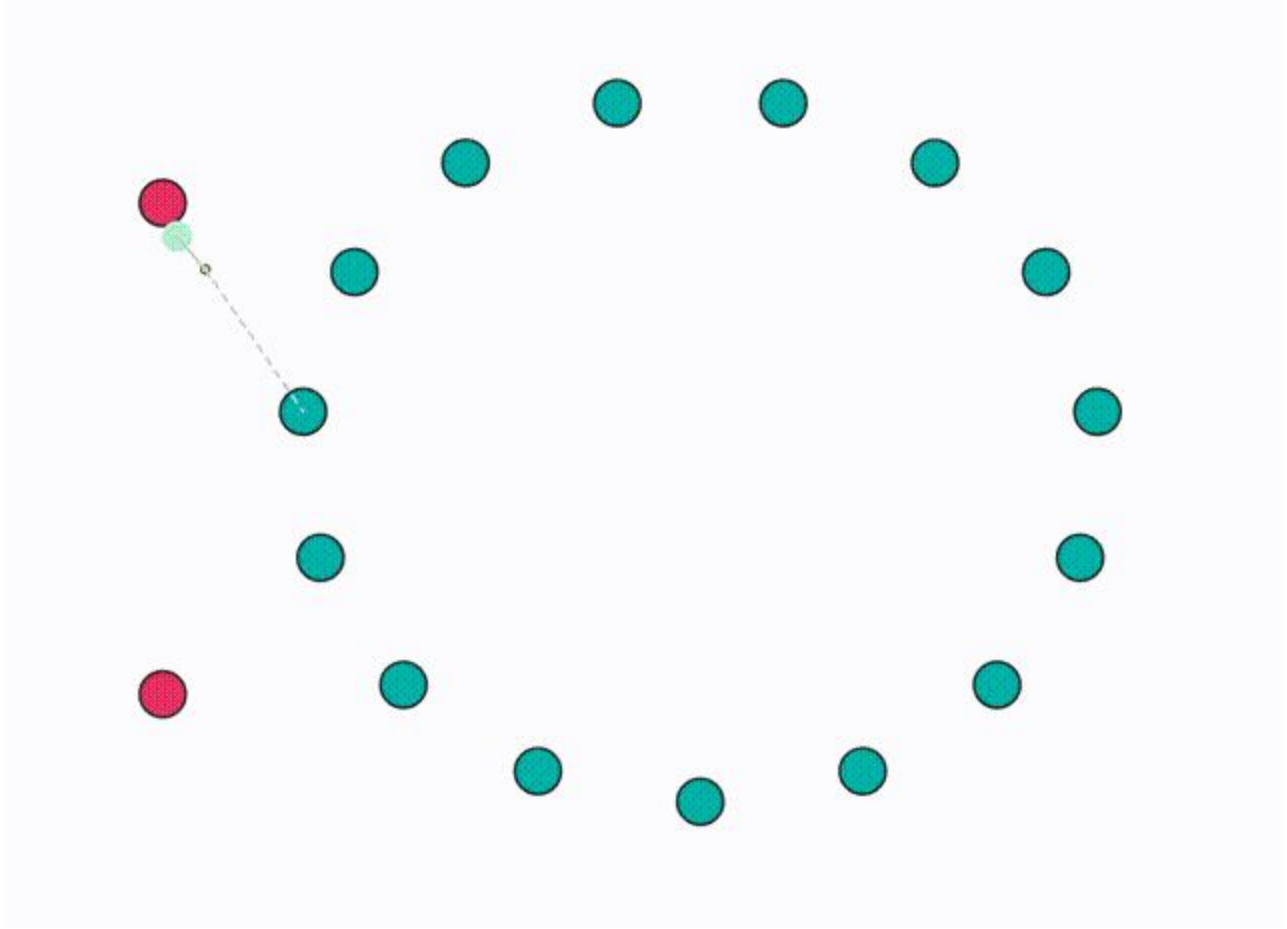


Figure 2: API for fault-tolerant log.

3. Paxos Refresh



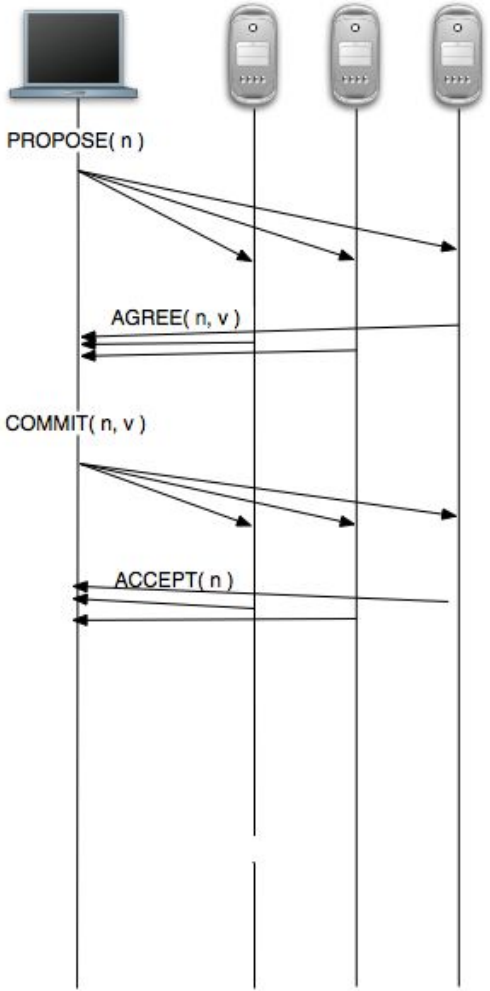
The part-time parliament

- Leslie Lamport 1980s
- Consensus algorithm that sacrifices liveness for correctness

Three Phases of Paxos

- Elect a replica to be the coordinator (distinguished proposer)
- Coordinator broadcasts a message with a selected value to the replicas asking them to accept, replicas can acknowledge or reject the message
- Once a majority acknowledge, consensus has been reached, and the coordinator broadcasts a commit message

Paxos: Correct Run



Ensures consensus can be reached on single value

- assigning ordering to coordinators
- restricting each coordinators choice of proposal value

Important notes

- Replicas must follow through on promises made in order to guarantee correctness
- Multiple coordinators can exist!

Optimizations: Multi-paxos

- Election phase can be omitted if the coordinator doesn't change
- Proposed values can be batched
- Lagging replicas can be caught up to leading replicas (catch-up protocol)

4. Algorithmic Challenges

A. Handling disk corruption

Problem: "When a replica’s disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm."

Solution: Two scenarios

- File changed: Checksums for each file to detect when a file changes
- File inaccessible: A marker left by new replicas in GFS after startup

When corruption is detected, replica goes into "catch-up" mode where it participates in paxos as a non-voting member.

B. Master Leases

Problem: "[R]ead operations cannot be served out of the master's copy of the data structure because it is possible that other replicas have elected another master and modified the data structure without notifying the old master."

Solution

- As long as the master has the lease, no other replica may submit a value to paxos
- All replicas implicitly grant a new lease to the master of the previous paxos instance
- The master maintains a shorter timeout for the lease than the replicas; clock drift protection

C. Epoch numbers

Problem: "[W]hen the master replica receives the request to the moment the request causes an update of the underlying database, the replica may have lost its master status... Chubby requires an incoming request to be aborted if mastership is lost..."

?????

"We solved this problem by introducing a global epoch number with the following semantics. Two requests for the epoch number at the master replica receive the same value iff that replica was master continuously for the time interval between the two requests."

C. Epoch numbers ...

Solution?:

- Global epoch number is stored as an entry in the database
- All database operations are made conditional on its value

D. Group membership

Problem: ?? "group membership problem"

Solution: "The details – though relatively minor – are subtle and beyond the scope of this paper."

E. Snapshots

Problem: The log is ever growing!

"This has two problems: it requires unbounded amounts of disk space; and perhaps worse, it may result in unbounded recovery time since a recovering replica has to replay a potentially long log before it has fully caught up with other replicas."

E. Snapshots ...

Solution

- Application layer can take a snapshot of the db
- The replicated log framework can truncate the log based on the snapshot contents
- The snapshot and truncation happen in an organized manner to guarantee no data loss
- Lagging replicas can catch-up via the snapshot when live logs are not available

F. Database transactions

Problem: Chubby application requires an atomic compare and swap semantics from the log of operations (paxos log)

Solution: “MultiOp” primitive

- list of tests called "guard"
- list of operations called "t_op" which get executed if guard evaluates -> true
- list of operations called "f_op" which get executed if guard evaluates -> false

5. Software Engineering Challenges

A. Effective algorithm expression

Problem: The algorithm is dense and the implementation code makes it complex

Solution: Build two explicit state machines and an "algorithm compiler"

B. Runtime consistency checking

Problem: The size of the project drastically increases the likelihood that inconsistencies will crop up in the code

Solution: periodic database log checksum comparisons between master and replicas

C. Testing

Problem: "it is unrealistic to prove a real system such as ours correct"

Solution: Increase robustness via in depth thorough testing!

Two test modes

- Safety: Consistency is verified but progress is not required
- Liveness: Both consistency and progress are required

C. Testing ...

Testing uses a random number generator as a seed for the test stage. If the tests fail, the seed is used to deterministically run the tests again with a debugger attached.

"In order to improve its bug yield, we started running this test on a farm of several hundred Google machines at a time. We found additional bugs, some of which took weeks of simulated execution time (at extremely high failure rates) to find."

"By their very nature, fault-tolerant systems try to mask problems. Thus they can mask bugs or configuration problems while insidiously lowering their own fault-tolerance."

D. Concurrency

Problem: Concurrent code is hard to repeatedly test deterministically

Solution: Start writing for single-threads and then expand to concurrency as needed

"In summary, we believe that we set ourselves the right goals for repeatability of executions by constraining concurrency. Unfortunately, as the product needs grew we were unable to adhere to these goals."

6. Unexpected Failures

"Our first release shipped with ten times the number of worker threads as the original Chubby system. We hoped this change would enable us to handle more requests. Unfortunately, under load, the worker threads ended up starving some other key threads and caused our system to time out frequently. This resulted in rapid master failover, followed by en-masse migrations of large numbers of clients to the new master which caused the new master to be overwhelmed, followed by additional master failovers, and so on."

7. Conclusion

Test	# workers	file size (bytes)	Paxos-Chubby (100MB DB)	3DB-Chubby (small database)	Comparison
Ops/s Throughput	1	5	91 ops/sec	75 ops/sec	1.2x
Ops/s Throughput	10	5	490 ops/sec	134 ops/sec	3.7x
Ops/s Throughput	20	5	640 ops/sec	178 ops/sec	3.6x
MB/s Throughput	1	8 KB	345 KB/s	172 KB/s	2x
MB/s Throughput	4	8 KB	777 - 949 KB/s	217 KB/s	3.6 - 4.4x
MB/s Throughput	1	32 KB	672 - 822 KB/s	338 KB/s	2.0 - 2.4x

Table 1: Comparing our system with 3DB (higher numbers are better).

"There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol."