# CASPaxos: Replicated State Machines without logs

Denis Rystsov
rystsov.denis@gmail.com

March 26, 2018

**Abstract**

CASPaxos is a replicated state machine (RSM) protocol, an extension of Synod. Unlike Raft and Multi-Paxos, it doesn't use leader election and log replication, thus avoiding associated complexity. Its symmetric peer-to-peer approach achieves optimal commit latency in wide-area networks and doesn't cause transient unavailability when any $\lfloor \frac{N-1}{2} \rfloor$ of $N$ nodes crash.

The lightweight nature of CASPaxos allows new combinations of RSMs in the designs of distributed systems. For example, a representation of key-value storage as a hashtable with independent RSM per key increases fault tolerance and improves performance on multi-core systems compared with a hashtable behind a single RSM.

This paper describes CASPaxos protocol, formally proves its safety properties, covers cluster membership change and evaluates the benefits of CASPaxos-based key-value storage.

## 1 Introduction

Multi-Paxos[1] and Raft[2] protocols allow a collection of nodes to work as a single state machine tolerating non-byzantine failures and network issues. The protocols guarantee safety in the presence of arbitrary node crashes, message loss and out-of-order delivery; and preserve liveness when at most $\lfloor \frac{N-1}{2} \rfloor$ of $N$ machines are down or disconnected.

RSMs and fault-tolerant strongly consistent (linearizable) storages are equivalent in a sense that one can be implemented via the other, and vice versa. So Multi-Paxos and Raft are widely used in the industry as a foundation of such databases as Chubby[3], Etcd[1], Spanner[4], etc.

Despite the wide adoption, there are a lot of indications that those protocols are complex. Diego Ongaro and John Ousterhout write in "In Search of an Understandable Consensus Algorithm"[2]:

> In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year

Google's engineers write about their experience of building a Paxos-based database in the "Paxos Made Live"[3] paper:

> Despite the existing literature in the field, building such a database proved to be non-trivial ... While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code ... There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system.

The complexity of the RSM protocols may explain the bugs in the replication layer of the established databases. Kyle Kingsbury made a comprehensive research[2] of the distributed consistent storages and found violations of linearizability in some version of almost every system he tested including MongoDB, Etcd, Consul, RethinkDB, VoltDB, and CockroachDB.

Besides complexity, those protocols have temporary availability problems when a leader crashes or is isolated. The "There Is More Consensus in Egalitarian Parliaments" paper[5] describes the negative implications of a leader-based system which are applicable both to Multi-Paxos and Raft:

> Traditional Paxos variants are sensitive to both long-term and transient load spikes and network delays that increase latency at the master ... this single-master optimization can harm availability: if the

[1] https://github.com/coreos/etcd
[2] https://aphyr.com/tags/jepsen

> master fails, the system cannot service requests until a new master is elected ... Multi-Paxos has high latency because the local replica must forward all commands to the stable leader.

**Contributions.** We present CASPaxos, a novel protocol for building RSM that avoids complexity and pitfalls of the log-based systems.

Multi-Paxos based system is a RSM built on top of a replicated log which treats every log entry as a command. The replicated log is composed of an array of Synod[1] (also known as Single Decree Paxos) instances. According to the Raft paper, its complexity comes from the composition rules:

> We hypothesize that Paxos' opaqueness derives from its choice of the single-decree subset as its foundation ... The composition rules for Multi-Paxos add significant additional complexity and subtlety.
>
> One reason is that there is no widely agreed upon algorithm for multi-Paxos. Lamport's descriptions are mostly about single-decree Paxos; he sketched possible approaches to multi-Paxos, but many details are missing. As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, discovers the difficulties in implementing it, and then develops a significantly different architecture ... real implementations are so different from Paxos that the proofs have little value

Instead of using Synod as a building block, CASPaxos extends it, so there is no composition and the associated complexity.

An experimental study[2] and the number of open source implementations[3] indicate that Raft succeeded in its goal to be understandable. However, its complexity is still comparable with Multi-Paxos: both systems[3][2] have several thousand of lines of code, both use concepts of leader election and leases, both are based on logs and require log compaction. CASPaxos is significantly simpler: it doesn't have those pieces and our implementation[4] is less than 500 lines of code.

Being just an extension of Synod, CASPaxos uses its symmetric peer-to-peer approach and automatically achieves the goals set in the EPaxos[5] paper: (1) optimal commit latency in the wide-area when tolerating one and two failures, under realistic conditions; (2) uniform load balancing across

[3] https://raft.github.io/#implementations
[4] https://github.com/gryadka/js

all replicas (thus achieving high throughput); and (3) graceful performance degradation when replicas are slow or crash.

**Verification**. The formal proof is given in the appendix A, Tobias Schottdorf and Greg Rogers independently model checked the protocol with TLA+[5], and the implementation was successfully tested with fault injection technique.

# 2 Protocols

We begin by briefly describing the Synod protocol from the perspective of master-master replication, followed by a step by step comparison with CAS-Paxos.

## 2.1 Synod

An implementation of the Synod protocol is an initializable-only-once distributed register. When several clients try to initialize it concurrently, the requests either prevent each other from continuing, or a single initialization succeeds. Once a client receives a confirmation, all the follow-up initializations must return the already chosen value.

The register belongs to the CP category of the CAP theorem[6], so it gives up availability when more than $\lfloor \frac{N-1}{2} \rfloor$ nodes are down but always preserves consistency (safety, linearizability).

Each node in the system plays the role of client, proposer or acceptor.

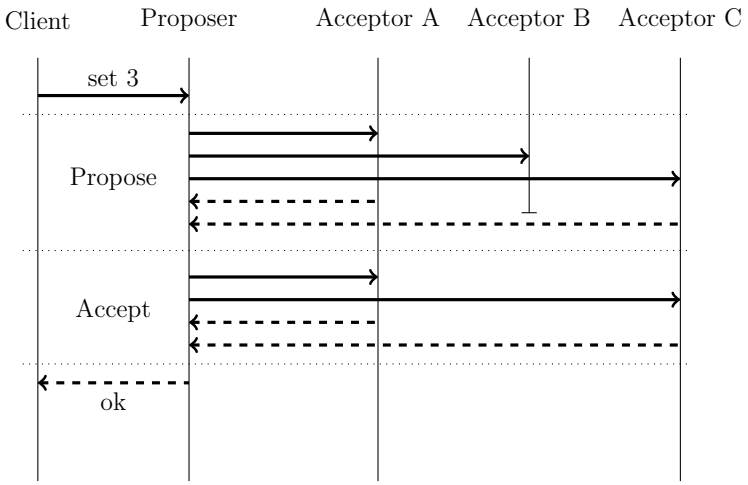
**Clients** initiate a request by communicating with a proposer; clients may be stateless, the system may have arbitrary numbers of clients.

**Proposers** perform the initialization by communicating with acceptors. Proposers keep minimal state needed to generate unique increasing update IDs (ballot numbers), the system may have arbitrary numbers of proposers.

**Acceptors** store the accepted value; the system should have $2F+1$ acceptors to tolerate $F$ failures.

[5] https://tschottdorf.github.io/single-decree-paxos-tla-compare-and-swap, https://medium.com/@grogepodge/tla-specification-for-gryadka-c80cd625944e

[6] https://en.wikipedia.org/wiki/CAP_theorem




It's convenient to use tuples as ballot numbers. To generate it a proposer combines its numerical ID with a local increasing counter: (counter, ID). To compare ballot tuples, we should compare the first component of the tuples and use ID only as a tiebreaker.

When a proposer receives a conflicting message from an acceptor, it should fast-forward its counter to avoid a conflict in the future.

## 2.2 CASPaxos

An implementation of CASPaxos is a rewritable distributed register. Clients change its value by submitting side-effect free functions which take the current state as an argument and yield new as a result. Out of the concurrent requests only one can succeed, once a client gets a confirmation of the change it's guaranteed that all future states are its descendants: there exists a chain of changes linking them together.

Just like with Synod, it's a CP-system, and it requires $2F + 1$ nodes to tolerate $F$ failures. Also, it uses the same roles: clients, proposers, and acceptors, and a very similar two-phase state transition mechanism.

Let's review the Synod and CASPaxos protocols step-by-step.

| **Synod** | **CASPaxos** |
|---|---|
| A client proposes the $val_0$ value to a proposer. | A client submits the $f$ change function to a proposer. |

| | |
|---|---|
| The proposer generates a ballot number, $B$, and sends "prepare" messages containing that number to the acceptors. | The proposer generates a ballot number, $B$, and sends "prepare" messages containing that number to the acceptors. |
| **An acceptor**<br><br>Returns a conflict if it already saw a greater ballot number.<br><br>Persists the ballot number as a promise and returns a confirmation either with an empty value (if it hasn't accepted any value yet) or with a tuple of an accepted value and its ballot number. | **An acceptor**<br><br>Returns a conflict if it already saw a greater ballot number.<br><br>Persists the ballot number as a promise and returns a confirmation either with an empty value (if it hasn't accepted any value yet) or with a tuple of an accepted value and its ballot number. |
| **The proposer**<br><br>Waits for the $F + 1$ confirmations<br><br>If they all contain the empty value, then the proposer defines the current state as $val_0$ otherwise it picks the value of the tuple with the highest ballot number.<br><br>Sends the current state along with the generated ballot number $B$ (an "accept" message) to the acceptors. | **The proposer**<br><br>Waits for the $F + 1$ confirmations<br><br>If they all contain the empty value, then the proposer defines the current state as $\emptyset$ otherwise it picks the value of the tuple with the highest ballot number.<br><br>Applies the $f$ function to the current state and sends the result, new state, along with the generated ballot number $B$ (an "accept" message) to the acceptors. |
| **An acceptor**<br><br>Returns a conflict if it already saw a greater ballot number.<br><br>Erases the promise, marks the received tuple (ballot number, value) as the accepted value and returns a confirmation | **An acceptor**<br><br>Returns a conflict if it already saw a greater ballot number.<br><br>Erases the promise, marks the received tuple (ballot number, value) as the accepted value and returns a confirmation |

| **The proposer** | **The proposer** |
|---|---|
| Waits for the $F+1$ confirmations | Waits for the $F+1$ confirmations. |
| Returns the current state the client. | Returns the new state to the client. |

As we see, the CASPaxos's state transition is almost identical to the Synod's initialization, and if we use

$$x \to \text{if } x = \emptyset \text{ then } val_0 \text{ else } x$$

as the change function then it indeed becomes identical.

We may use the following change functions to turn CASPaxos into a distributed compare and set register:

- To **initialize** a register with $val_0$ value

$$x \to \text{if } x = \emptyset \text{ then } (0, val_0) \text{ else } x$$

- To **update** a register to value $val_1$ if the current version is 5

$$x \to \text{if } x = (5, *) \text{ then } (6, val_1) \text{ else } x$$

- To **read** a value

$$x \to x$$

With this specialization, the protocol is almost indistinguishable from Bizur[6].

### 2.2.1 One-round trip optimization

Since the prepare phase doesn't depend on the change function, it's possible to piggyback the next prepare message on the current accept message to reduce the number of round trips from two to one.

In this case, a proposer caches the last written value, and the clients should use that proposer to initiate the state transition to benefit from the optimization.

### 2.2.2 Low-latency and high-throughput consensus across WAN deployments

WPaxos[7] paper describes how to achieve low-latency and high-throughput consensus across wide area network through object stealing. It leverages the flexible quorums[8] idea to cut WAN communication costs. Since CASPaxos is an extension of Synod and supports FPaxos (see the proof in the appendix B), it can benefit the idea too.

## 2.3 Cluster membership change

Cluster membership change is a process of changing the set of nodes executing a distributed system without violating safely and liveness properties. It's crucial to have this process because it solves two problems:

1. *How to adjust fault tolerance properties of a system.* With time the fault tolerant requirements may change. Since a CASPaxos-based system of size $N$ tolerates up to $\lfloor\frac{N-1}{2}\rfloor$ crashes, a way to increase/decrease size of a cluster is also a way to increase/decrease resiliency of the system.

2. *How to replace permanently failed nodes.* CASPaxos tolerates transient failures, but the hardware tends to break, so without a replacement eventually, more than $\lfloor\frac{N-1}{2}\rfloor$ nodes crash, and the system becomes unavailable. A replacement of a failed node in the $N$ nodes cluster can be modeled as a shrinkage followed by an expansion.

The process of membership change is based on Raft's idea of joint consensus where two different configurations overlap during transitions. It allows the cluster to continue operating normally during the configuration changes.

The proof of applicability of this idea to CASPaxos is based on two observations:

- **Flexible quorums** It has been observed before that in a Paxos-based system the only requirement for the "prepare" and "accept" quorums is the intersection [9][10][8]. For example, if the cluster size is 4, then we may require 2 confirmations during the "prepare" phase and 3 during the "accept" phase.

- **Network equivalence** If a change in the behavior of a Paxos-based system can be explained by delaying or omitting the messages between

the nodes, then the change doesn't affect consistency because Paxos tolerates the interventions of such kind. It gives freedom in changing the system as long as the change can be modeled as a message filter on top of the unmodified system.

### 2.3.1 Expansion of a cluster with an odd number of nodes

The protocol for changing the set of acceptors from $A_1 \cdots A_{2F+1}$ to $A_1 \cdots A_{2F+2}$:

1. Turn on the $A_{2F+2}$ acceptor.
2. Connect to each proposer and update its configuration to send the "accept" messages to the $A_1 \cdots A_{2F+2}$ set of acceptors and to require $F+2$ confirmations during the "accept" phase.
3. Pick any proposer and execute the identity state transition function $x \to x$.
4. Connect to each proposer and update its configuration to send "prepare" messages to the $A_1 \cdots A_{2F+2}$ set of acceptors and to require $F+2$ confirmations.

From the perspective of the $2F+1$ nodes cluster, the second step can be explained with the network equivalence principle, so the system keeps being correct. When all proposers are modified the system also works as a $2F+2$ nodes cluster with $F+1$ "prepare" quorum and $F+2$ "accept" quorum.

After the read operation finishes the state of the cluster becomes valid from the $F+2$ perspective, so we can forget about the $F+1$ interpretation. The last step switches the system from the reduced "prepare" quorum to the regular.

The same sequence executed in the reverse order shrinks cluster with an even number of nodes.

### 2.3.2 Expansion of a cluster with an even number of nodes

The $A_1 \cdots A_{2F+2}$ to $A_1 \cdots A_{2F+3}$ extension protocal is more straightforward because we can treat a $2F+2$ nodes cluster as a $2F+3$ nodes cluster where one node had been down from the beginning:

1. Connect to each proposer and update its configuration to send the prepare & accept messages to the $A_1 \cdots A_{2f+3}$ set of acceptors.

2. Turn on the $A_{2f+3}$ acceptor.

It's important to notice that the procedure works based on the assumption that the $2f + 3^{\text{th}}$ node has always been down. If a cluster gets into even configuration from an odd configuration, then it's necessary to execute identity state transition (re-scan) before the extension to avoid data loss.

Otherwise, it's possible to sequentially replace every acceptor with an empty acceptor, lose all data and violate linearizability.

### 2.3.3 Optimization

In a key-value storage implemented as an array of independent labeled CASPaxos instances, we need to perform the 3 step for each instance (key). It results in a rescan of all record. If the storage has $K$ keys then during the $A_1 \cdots A_{2F+1}$ to $A_1 \cdots A_{2F+2}$ transition the rescan moves $K(2F + 3)$ records.

The goal of the identity state transition is to make the state of the cluster valid from the $F + 2$ perspective. The alternative way to reach this state is to replicate a majority of $A_1 \cdots A_{2F+1}$ nodes for any moment after the 2 step into $A_{2F+2}$ resolving conflicts by choosing an accepted value with higher ballot number. Thus reducing the rescan cost from $K(2F + 3)$ to $K(F + 1)$.

A background catch-up process keeping acceptors in sync up to some recent moment may reduce the cost further to $(K - k) + k(F + 1)$ where $k$ is the number of updated keys since the last moment of sync.

### 2.3.4 Changing the number of proposers

Consistency and availability properties don't depend on the number of proposers so that they can be added and removed at any time. The only caveat is the procedures like shrinkage-expansion of acceptors and deletion (3.1) which need to update all proposers as one of the steps. Fortunately, those steps are idempotent so we so we can bring a proposer down, update the list of all proposers and on the next attempt the steps succeed.

An algorithm to remove a proposer:

1. Turn off the proposer.
2. Update the list of proposers of the GC process.
3. Update the list of proposers of the process controlling the shrinkage-expansion of acceptors.

An algorithm to add a proposer:

1. Update the list of proposers of the process controlling the shrinkage-expansion of acceptors.
2. Update the list of proposers of the GC process.
3. Turn on the proposer.

# 3 A CASPaxos-based key-value storage

The lightweight nature of CASPaxos opens new simple ways for designing distributed systems with complex behavior. In this section, we'll discuss a CASPaxos-based design for a key-value storage and compare a research prototype with established databases.

The Raft paper[2] acknowledges that EPaxos[5] can achieve higher performance than Raft by using a leaderless approach and exploiting commutativity in state machine commands. A key-value storage with independent operations between keys looks like a good case for the EPaxos protocol. However, it adds significant complexity.

Alternatively, instead of putting the whole key-value storage under a single RSM and using the commutativity of the commands, we can use the lightweight nature of CASPaxos to run a RSM per key achieving uniform load balancing across all replicas (thus higher throughput).

Gryadka[7] is a prototype of a distributed key-value storage which uses that approach.

## 3.1 How to delete a record

CASPaxos supports only update (change) operation so to delete a value a client should update a register with an empty value (a tombstone). The downside of this approach is the space inefficiency: even when the value is empty, the system still spends space to maintain information about the register.

The straightforward removal of this information may introduce consistency issues. Consider the following state of the acceptors.

[7] https://github.com/gryadka/js

| | Promise | Ballot | State |
|---|---|---|---|
| Acceptor A | | 2 | 42 |
| Acceptor B | | 3 | $\emptyset$ |
| Acceptor C | | 3 | $\emptyset$ |

According to the CASPaxos protocol, a read operation (implemented as $x \to x$ change function) should return $\emptyset$. However, if we decide to remove all information associated with the register and the read request hits the system during the process when the data on acceptors B and C have already gone then the outcome is 42 which violates linearizability.

An increasing of the "accept" quorum to $2F + 1$ on writing an empty value before the removal solves the problem, but it makes the system less available since it's impossible to remove a register when at least one node is down.

A multi-step removal process fixes this problem.

1. On a delete request, a proposer writes a tombstone with regular $F + 1$ "accept" quorum, schedules a garbage collection operation and confirms the request to a client.

2. The garbage collection operation (in the background):

   (a) Replicates an empty value to all nodes by executing the identity transform with max quorum size $(2F + 1)$.

   (b) Connects to each proposer, invalidates its cache associated with the removing key (see one-round trip optimization 2.2.1), fast-forwards its counter to be greater than the tombstone's ballot number and increments proposer's age.

   (c) Connects to each acceptor and asks it to reject messages from proposers if their age is younger than the corresponding age from the previous step.

   (d) Removes the register from each acceptor if it contains the tombstone from the 2a step.

Each step of the GC process is idempotent so if any acceptor or proposer is down the process reschedules itself.

Invalidation of the proposer's caches and the age check are necessary to eliminate the lost delete anomaly, a situation when a message delayed by a

channel (or an accept message corresponding to a change of the cached value) revives a value without a causal link to the deletion event.

The update of the counters is necessary to avoid the lost update anomaly which may happen when a concurrently updated value has lesser ballot number than the tombstone's ballot number, and a reader prioritizes the tombstone over the new value.

To make the age check possible, proposers should include their age into every message they send, and acceptors should persist age per proposer set by GC process.

## 3.2 Low latency

The following behavior helps CASPaxos achieve low latency:

- CASPaxos isn't a leader-based protocol so a proposer should not forward all requests to a specific node to start executing them.
- Requests affecting different key-value pairs do not interfere.
- It uses 1RTT when the requests affecting the same key land on the same proposer.
- No acceptor is special, so a proposer ignores slow acceptors and proceeds as soon as quorum is reached.
- An ability to use user-defined functions as state transition functions reduces two steps transition process (read, modify, write) into one step process.

We can't get the same behavior from the Multi-Paxos or Raft based systems, so their latency is higher. The EPaxos paper emphasizes the following latency issue of the leader-based consensus protocols:

> Multi-Paxos has high latency because the local replica must forward all commands to the stable leader . . . when performing geo-replication, clients incur additional latency for communicating with a remote master

The Bizur paper[6] focuses on another log related latency degradation:

> A single slow operation will increase the latency of all succeeding operations, until the slow operation is committed. For example, a network packet drop will affect multiple ongoing operations (instead of affecting just the operation within the dropped packet)

Let's compare Gryadka, an experimental CASPaxos-based key-value storage, with the established storages Etcd and MongoDB to check how this a priori reasoning matches the real world.

All storages were tested in the same environment. Gryadka, Etcd, and MongoDB were using three DS4_V2 nodes configuration deployed over WAN in the Azure's[8] datacenters in the "West US 2", "West Central US" and "Southeast Asia" regions.

Each node has a colocated client which in one thread in a loop was reading a value, incrementing and writing it back. All clients used their keys to avoid collisions. During the experiment latency (average duration of read-modify-write operation) was measured per each client (region).

| | Latency | | |
|---|---|---|---|
| | MongoDB (3.6.1) | Etcd (3.2.13) | Gryadka (1.61.8) |
| West US 2 | 1086 ms | 679 ms | 47 ms |
| West Central US | 1168 ms | 718 ms | 47 ms |
| Southeast Asia | 739 ms | 339 ms | 356 ms |

The result matches our expectation especially if we take into account delay between datacenters and the leader/leaderless nature of MongoDB, Etcd, and Gryadka.

| | | RTT |
|---|---|---|
| West US 2 | West Central US | 21.8 ms |
| West US 2 | Southeast Asia | 169 ms |
| West Central US | Southeast Asia | 189.2 ms |

It happened that leaders of MongoDB and Etcd were in the "Southeast Asia" region so to execute an operation the "West US 2" node needs additional round trip to forward a request to the leader and to receive a response (169 ms). Then the leader needs to write the change to the majority of nodes and get confirmations (0 ms and 169 ms). Since the iteration consists of reading and writing operations, in total "West US 2" node requires at least $676\text{ms} = 2 \cdot (169\text{ms} + 169\text{ms})$. For the "West Central US" node the estimated latency is $716.4\text{ms} = 2 \cdot (169\text{ms} + 189.2\text{ms})$, for "Southeast Asia" it's $338\text{ms} = 2 \cdot 169\text{ms}$.

[8] https://azure.microsoft.com

Gryadka doesn't forward requests so the corresponding estimated latencies are $43.6\text{ms} = 2 \cdot 21.8\text{ms}$, $43.6\text{ms} = 2 \cdot 21.8\text{ms}$ and $338\text{ms} = 2 \cdot 169\text{ms}$.

Network fluctuations and storage implementation details may explain the difference between estimated and measured latencies.

As we see, the underlying consensus protocol plays an essential role in the performance of the system.

### 3.3 Fault-tolerance

The EPaxos paper explains how the leader-based consensus protocols lead to cluster-wide unavailability when a leader crashes or is isolated from the cluster:

> With Multi-Paxos, or any variant that relies on a stable leader, a leader failure prevents the system from processing client requests until a new leader is elected. Although clients could direct their requests to another replica (after they time out), a replica will usually not try to become the new leader immediately

CASPaxos doesn't suffer from this behavior because all of its nodes of the same role are homogeneous and when any of them is isolated it doesn't affect processes running against the others. An experimental study[9] of distributed consistent databases with default settings during a leader isolation accident supports this a priori reasoning - all systems but Gryadka has a non-zero unavailability window.

Please avoid comparing the systems by the unavailability window because it's a configuration parameter depending on RTT between nodes[10] and different databases have different defaults.

## 4 Comparison with Related Work

Raft[2] is similar to CASPaxos in its origin: both were created as an attempt to overcome the complexity of Multi-Paxos. Raft uses the same concepts as Multi-Paxos like leader election and log replication but rearranges them differently with a focus on understandability. CASPaxos chooses a different foundation and has fewer moving parts. As a result, its implementation is $\frac{1}{4}$

[9] https://github.com/rystsov/perseus

[10] https://coreos.com/etcd/docs/latest/tuning.html

| Database | Version | Protocol | Unavailability |
|---|---|---|---|
| Gryadka | 1.61.8 | CASPaxos | 0s |
| CockroachDB | 1.1.3 | MultiRaft | 7s |
| Consul | 1.0.2 | Raft | 14s |
| Etcd | 3.2.13 | Raft | 1s |
| RethinkDB | 2.3.6 | Raft | 17s |
| Riak | 2.2.3 | Vertical Paxos | 8s |
| TiDB | 1.1.0 | MultiRaft | 15s |

of Raft's regarding the lines of code and the symmetric peer-to-peer approach allows it to experience isolation of a node without impacting clients.

Bizur[6] is a protocol for building key-value storages; it relates to CASPaxos the same way as Synod does. CASPaxos is a replicated state machine which allows clients to submit functions to change its state. By choosing one set of functions, we specialize it to be a write-once register, Synod; by choosing another set, we get a rewritable register, Bizur. However, the Bizur paper doesn't specify how to remove values (buckets) other than by creating tombstones.

EPaxos[5] is a leaderless variant of Multi-Paxos which allows concurrent execution of non-interfering commands. CASPaxos doesn't allow concurrent state transition, but in some cases, it provides comparable functionality with simpler design. For example, key-value storage with independent operations between keys can be modeled as a single EPaxos-based RSM or as storage with a CASPaxos-based RSM per key. Both systems achieve optimal commit latency, uniform load balancing across all replicas and graceful performance degradation when replicas are slow or crash.

# 5 Conclusion

Despite log-based Paxos-like consensus protocols being complex and having latency issues during failures, they find wide adoption in the industry as a foundation of distributed databases. However many of the applications of the protocols are implementations of master-master replicated key-value storages.

CASPaxos is the consensus protocol which overcomes the complexity of log-based solutions like Multi-Paxos and Raft and uses symmetric peer to

peer approach to avoid the latency issues. The lightweight nature of CASPaxos allows to use it as a foundation of key-value storage with simple design and better resiliency guarantees.

The indirect contributions of our work are 1) the proof of safety properties which is also applicable to the other variants of Paxos such as FPaxos, and 2) the network equivalence principle which automatically proves the preservation of properties during a transition from one configuration to another.

**Acknowledgements**. We thank Tobias Schottdorf and Greg Rogers who wrote TLA+ specs for CASPaxos, and Ezra Hoch who found a flaw in the first version of the deletion process 3.1.

# References

[1] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[2] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[3] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[4] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[5] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[6] Ezra N Hoch, Yaniv Ben-Yehuda, Noam Lewis, and Avi Vigder. Bizur: A key-value consensus algorithm for scalable file-systems. 2017, 1702.04242.

[7] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Ruling the archipelago with fast consensus. 2017, 1703.08905.

[8] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. 2016, 1608.06696.

[9] Butler Lampson. The abcd's of paxos. In *PODC*, volume 1, page 13, 2001.

[10] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313. ACM, 2009.

# Appendices

## A Proof

**Theorem 1.** *We want to prove that for any two acknowledged changes one is always a descendant of another.*

*Let $\bar{E}^2$ represent a set of acknowledged events (happen on proposers when they receive at least $F+1$ confirmations during the "accept" phase), and $\rightarrow$ represent the "is a descendant" relation; then we want to demonstrate that.*

$$\forall x, y \in \bar{E}^2 \ : \ x \rightarrow y \vee y \rightarrow x \tag{1}$$

**Definition. ("is a descendant" relation)** What does it mean that one event is a descendant of another? Informally it means that there is a chain of state transitions leading from the state acknowledged in the initial event to the state acknowledged in the final event. Let's formalize it. We start by defining this relation between the successfully accepted message events and then expand it to $\bar{E}^2$. Accepted message events happen on acceptors, for each acknowledged message event there are at least $F+1$ accepted message events, the set of such events is denoted as $\ddot{E}^2$.

By definition of CASPaxos, any accepted state is a function of previously accepted state, so

$$\forall x \in \ddot{E}^2 \quad \exists! f \quad \exists! y \in \ddot{E}^2 : s(x) = f(s(y)) \tag{2}$$

Where $s(x)$ is a state accepted by an acceptor resulting in event $x$. When 2 holds for $x$ and $y$ we write $y \sim x$ and $y = I^{-1}(x)$. Now we can define "is a descendant" relation between $\ddot{E}^2$ events as:

$$\forall x \in \ddot{E}^2 \, \forall x \in \ddot{E}^2 \ : \ x \rightarrow y \equiv x \sim y \vee (\exists z \in \ddot{E}^2 \ : \ x \rightarrow z \wedge z \rightarrow y) \tag{3}$$

Let's define $x.w$ for $x \in \bar{E}^2$ as accept message events which correspond to the acknowledged change $x$. By definition, the following properties hold:

1. $\forall x \in \bar{E}^2 \ x.w \subset \ddot{E}^2$
2. $\forall x \in \bar{E}^2 \ |x.w| >= F+1$ (we require a quorum of confirmations before acknowledging a change)

3. $\forall x \in \bar{E}^2 \; \forall y \in x.w \;\; : \;\; s(x) = s(y)$ (accepted state and acknowledged state is the same)

The third property allows continuing "is a descendant" relation on $\bar{E}^2$:

$$\forall x \in \bar{E}^2 \; \forall y \in \bar{E}^2 \;\; : \;\; x \to y \equiv (\forall a \in x.w \; \forall b \in y.w \; a \to b) \tag{4}$$

**Lemma 2.** *The following statement proves the theorem 1.*

$$\forall x \in \bar{E}^2 \; \forall y \in \ddot{E}^2 \;\; : \;\; x.b < y.b \implies x.b \leq I^{-1}(y).b \tag{5}$$

*Where $x.b$ means a ballot number of an acknowledged or an accepted event.*

*Proof.* Let $z_0 := y$ and $z_{n+1} := I^{-1}(z_n)$. By definition, ballot numbers only increase: $z_{n+1}.b < z_n.b$, so we can use mathematical induction and 5 guarantees that $\exists k \;\; : \;\; z_k.b = x.b$ meaning $s(z_k) = s(x)$. Since $z_{k+1} \sim z_k$ we proved the following statement:

$$\forall x \in \bar{E}^2 \; \forall y \in \ddot{E}^2 \;\; : \;\; x.b < y.b \implies x \to y \tag{6}$$

Since $\forall y \in \bar{E}^2 \; \forall z \in y.w \;\; : \;\; y.b = z.b \wedge s(y) = s(z)$ then 6 implies

$$\forall x \in \bar{E}^2 \; \forall y \in \bar{E}^2 \;\; : \;\; x.b < y.b \implies x \to y \tag{7}$$

By definition, $\forall x \in \bar{E}^2 \; \forall y \in \bar{E}^2 \;\; : \;\; x.b < y.b \vee y.b < x.b$ so the latter means

$$\forall x \in \bar{E}^2 \; \forall y \in \bar{E}^2 \;\; : \;\; x \to y \vee y \to x \tag{8}$$

Which proves the theorem 1. □

Before proving the 5 let's define $\ddot{E}^1$ as a set of promised events (happen on acceptors during the prepare phase) and $x.r$ for $x \in \ddot{E}^2$ as promised events corresponding to the acknowledged change $x$. By definition, the following properties hold:

1. $\forall x \in \ddot{E}^2 \; x.r \subset \ddot{E}^1$
2. $\forall x \in \ddot{E}^2 \; |x.r| >= F + 1$ (we require a quorum of confirmations before staring the accept phase)

.

**Theorem 3.**

$$\forall x \in \bar{E}^2 \, \forall y \in \ddot{E}^2 \; : \; x.b < y.b \implies x.b \leq I^{-1}(y).b$$

*Proof.* Let

$$N = \{z.node \mid z \in x.w\} \cap \{z.node \mid z \in y.r\}$$

$N$ isn't empty because "prepare" quorum intersects with "accept" quorum. Let $n \in N$, and $w \equiv x.w|_n$ and $u \equiv y.r|_n$ are accepted and promised events on node $n$. By definition,

$$w.b < u.b \tag{9}$$

It's true because event $w$ happened before $u$ in $n$'s timeframe since an acceptor doesn't accept messages with lesser ballot numbers than they already saw and $w.b = x.b < y.b = u.b$.

Let $P \equiv \{x \mid x \in \ddot{E}^1 \wedge x.node = n\}$ and each event in $P$ have the following structure:

| $x \in P$ | |
|---|---|
| $x.ts$ | local time on node $x.node$ |
| $x.b$ | promise, a ballot number of last prepare message |
| $x.ret.b$ | a ballot number of the last accepted state |
| $x.ret.s$ | the last accepted state |
| $x.node$ | a node where $x$ was emitted |
| $s(x)$ | is equal to $x.ret.s$ |

Let $A \equiv \{x \mid x \in \ddot{E}^2 \wedge x.node = n\}$ and each event's structure is:

| $x \in A$ | |
|---|---|
| $x.ts$ | local time on node $x.node$ |
| $x.b$ | a ballot number of the last accepted state |
| $x.s$ | the last accepted state |
| $x.node$ | a node where $x$ was emitted |
| $s(x)$ | is equal to $x.s$ |

For $x$ in $P$, $x.ret$ is the latest accepted state, let's write it formally.

$$\forall k \in P \quad k.ret.b = \max\{l.b \mid l \in A \wedge l.ts < k.ts\} \tag{10}$$

Since $w.b < u.b$, $w \in A$ and $u \in P$

$$w \in \{z \in A, z.ts < u.ts\} \tag{11}$$

With combination with 10 it implies:

$$x.b = w.b \leq \max\{z.b \in A, z.ts < u.ts\} = u.ret.b \tag{12}$$

By definition a proposer picks the value with max ballot number as the current state out of quorum of promise confirmations, so:

$$I^{-1}(y).b = \max\{z.ret.b | z \in y.r\} \tag{13}$$

Combining with 12 we get:

$$\begin{aligned} x.b = w.b \leq \max\{z \in A, z.ts < u.ts\} = \\ = u.ret.b \leq \max\{z.ret.b | z \in y.r\} = I^{-1}(y).b \end{aligned} \tag{14}$$

Which proves $x.b \leq I^{-1}(y).b$.

□

# B FPaxos

The proof of CASPaxos A doesn't use the size of the promise/accept quorums, it depends only on their intersection, so the same proof proves FPaxos too.