# CASPaxos: Replicated State Machines without logs

**Denis Rystsov**
Microsoft
derystso@microsoft.com

## Abstract

CASPaxos is a wait-free, linearizable, multi-writer multi-reader register in unreliable, asynchronous networks supporting arbitrary update operations including compare-and-set (CAS). The register acts as a replicated state machine providing an interface for changing its value by applying an arbitrary user-provided function (a command). Unlike Multi-Paxos and Raft which replicate the log of commands, CASPaxos replicates state, thus avoiding associated complexity, reducing write amplification, increasing concurrency of disk operations and hardware utilization.

The paper describes CASPaxos, proves its safety properties and evaluates the characteristics of a CASPaxos-based prototype of key-value storage.

**2012 ACM Subject Classification** Information systems → Distributed storage

**Keywords and phrases** atomic register, linearizability, paxos, consensus, wait-free

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

**Related Version** https://arxiv.org/abs/1802.07000

**Supplement Material** https://github.com/gryadka/js

**Acknowledgements** We thank Tobias Schottdorf and Greg Rogers for writing TLA+ specs for CASPaxos, and Ezra Hoch, Peter Bourgon, Lisa Kosiachenko, Reuben Bond and Murat Demirbas for valuable comments and suggestions.

## 1 Introduction

Replicated state machine (RSM) protocols allow a collection of nodes to work as a state machine tolerating non-byzantine node failures and communication problems. 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. This explains why Multi-Paxos[5] and Raft[7] are widely used in the industry as a foundation of distributed systems (e.g. Chubby[2], Etcd[1], Spanner[3]).

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"[7]:

> 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

[1] https://github.com/coreos/etcd


> 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"[2] 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.

Besides complexity, those protocols are subject to temporary loss of availability. The "There Is More Consensus in Egalitarian Parliaments" paper[6] 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 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 leaderless protocol for building RSM that avoids complexity associated with Multi-Paxos/Raft and availability implementations of having a leader and reduces write amplification.

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[5] (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

The main idea of CASPaxos is to replicate state instead of the log of commands. We achieve it by extending Synod instead of using it as a building block. As a consequence, there is no composition and the associated complexity. It allows our implementation[2] to be 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[6] paper: (1) optimal commit latency in the wide-area when tolerating failures under realistic conditions; (2) uniform load balancing across all replicas (thus achieving high throughput); and (3) graceful performance degradation when replicas are slow or crash.

---

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

## 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 CASPaxos.

### 2.1 Synod

An implementation of the Synod protocol is an initializable-only-once distributed register. When several clients try to initialize it concurrently - at most one client 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[3], gives up availability when more than $\lfloor \frac{N-1}{2} \rfloor$ nodes are down and always preserves consistency (linearizability).

Each node in the system plays the role either 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.

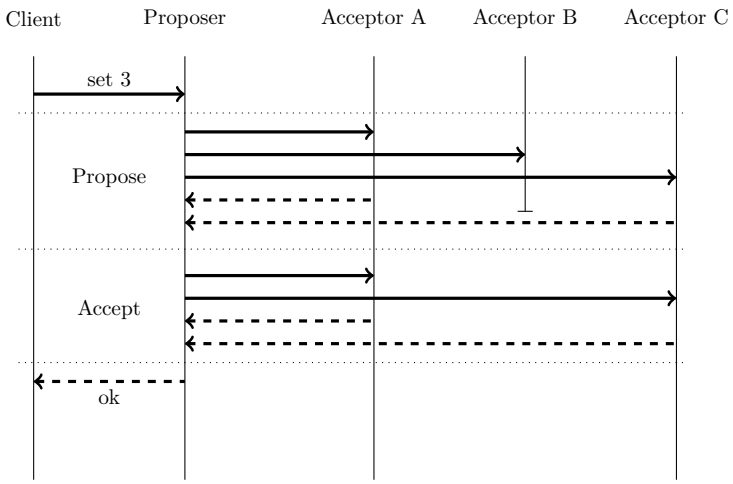


**Figure 1** Synod sequential diagram

It's convenient to use tuples as ballot numbers. To generate it a proposer combines its comparable 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 update its counter to avoid a conflict in the future.

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

## 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. | **An acceptor**<br><br>Returns a conflict if it already saw a greater ballot number. |

| Erases the promise, marks the received tuple (ballot number, value) as the accepted value and returns a confirmation | 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.

By choosing a different set of change functions, we can turn CASPaxos into a distributed register and atomically change its value with arbitrary user-defined function in one operation. For example, the following change functions implement a register with CAS.

- 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

$$x \to x$$

### 2.2.1 One-round trip optimization

Since the prepare phase doesn't depend on the change function, the next prepare message can piggyback 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. The optimization doesn't compromise safety if a client sends a request to another proposer.

## 3 Correctness

The protocol has a formal proof and has been model checked with TLA+. Let's start with the proof.

## 3.1 Proof

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

We'll do it by reasoning about the four types of the events:

- $e \in \ddot{E}^1$ - an acceptor received a "prepare" message and replied with "promise"
- $e \in \bar{E}^1$ - a proposer sent an "accept" message
- $e \in \ddot{E}^2$ - an acceptor accepted a new value
- $e \in \bar{E}^2$ - a proposer received a majority of confirmations

Let's review the structure of the events.

| $e \in \ddot{E}^1$ - promised event | |
|---|---|
| $e.ts$ | acceptor's local time |
| $e.b$ | ballot number of current change round |
| $e.ret.b$ | a ballot number of the last accepted value |
| $e.ret.s$ | the last accepted value |
| $e.node$ | acceptor's id |
| $s(e)$ | is equal to $e.ret.s$ |

| $e \in \bar{E}^1$ | |
|---|---|
| $e.ts$ | proposer's local time |
| $e.b$ | ballot number of current change round |
| $e.s$ | new value, result of a change function applied to a value corresponding to a promise with maximum ballot number |
| $s(e)$ | is equal to $e.s$ |

| $e \in \ddot{E}^2$ - accepted event | |
|---|---|
| $e.ts$ | acceptor's local time |
| $e.b$ | ballot number of current change round |
| $e.s$ | accepted new value |
| $e.r$ | $\{x \mid x \in \ddot{E}^1 \wedge x.b = e.b\}$ |
| $e.node$ | acceptor's id |
| $s(e)$ | is equal to $e.s$ |

| $e \in \bar{E}^2$ - acknowledged event | |
|---|---|
| $e.ts$ | proposer's local time |
| $e.b$ | ballot number of current change round |
| $e.s$ | accepted new value |
| $e.w$ | $\{x \mid x \in \ddot{E}^2 \wedge x.b = e.b\}$ |
| $e.node$ | proposer's id |
| $s(e)$ | is equal to $e.s$ |

**Figure 2** Structure of the events

We want to demonstrate that

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

where $\to$ represents the "is a descendant" relation.

▶ **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 on $\ddot{E}^2$ and then expand it to $\bar{E}^2$.

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

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

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 on $\ddot{E}^2$ events as:

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

We can use that $\forall x \in \bar{E}^2 \; \forall y \in x.w \; : \; s(x) = s(y)$ is true (by definition) and continue "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 1.** *The following statement proves that any two acknowledged changes one is always a descendant of another*

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

**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

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

◀

▶ **Theorem 2.**

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

**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 $n \in N$, $u \equiv y.r|_n$ is a promised event happened on node $n$ and $w \equiv x.w|_n$ is an accepted event. By definition, $y.b = u.b$, $w.b = x.b$ and $x.b < y.b$ so

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

Since an acceptor doesn't accept messages with lesser ballot numbers than they already saw, the latter means $w.ts < u.ts$

Let $P \equiv \{x \mid x \in \ddot{E}^1 \wedge x.node = n\}$ and $A \equiv \{x \mid x \in \ddot{E}^2 \wedge x.node = n\}$. Then for each $x \in P$, $x.ret.b$ is the ballot number of the latest accepted state, formally it means that:

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

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

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

With combination with 10 it implies:

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

By definition, a proposer picks a value out of majority of promise confirmations corresponding to the maximum ballot number, so:

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

Combining with 12 we get:

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

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

◀

## 3.2 Formal specification

Tobias Schottdorf and Greg Rogers independently model checked the protocol with TLA+[4].

# 4 A CASPaxos-based key-value storage

The lightweight nature of CASPaxos creates new 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 Etcd, MongoDB and other distributed databases.

Instead of designing a key-value storage as a single RSM we represent it as a set of labeled CASPaxos instances. Gryadka[5] is a prototype implementing this idea.

## 4.1 Low latency

The following properties of CASPaxos help achieve low latency:

- It 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.

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

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

- 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 compared Gryadka with Etcd and MongoDB to check it. 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[6] 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 |

The 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 $338\text{ms} = 2 \cdot 169\text{ms}$.

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 minor difference between estimated and measured latencies.

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

## 4.2 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:

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

> 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 so when any of them is isolated it doesn't affect processes running on the other nodes. An experimental study[7] 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 complete unavailability window (all client's operations halt).

| Database | Version | Protocol | Unavailability |
|---|---|---|---|
| Gryadka | 1.61.8 | CASPaxos | $< 1$s |
| 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 |

## 5 Comparison with Related Work

**Consensus.** CASPaxos and Raft/Multi-Paxos have different trade-offs. CASPaxos replicates state on each change request so it's impractical for RSM with a heavy monolithic state. The absence of logs eliminates artificial synchronization and reduces write amplification. Independent registers allow batching and out of order writing achieving high concurrency of disk operations thus better hardware utilization. Besides that, the absence of leader increases availability and reduces latency.

**Registers.** Protocols of atomic distributed registers, described in "Sharing memory robustly in message-passing systems"[1] and "On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory"[4] papers, has similar structure to CASPaxos. But they don't provide conditional writes or other concurrency control primitives which makes them impractical for concurrent environments and makes it's imposible to support client-side transactions.

## 6 Conclusion

We demonstrated that CASPaxos is a simple RSM protocol without availability implications of having a leader with high concurrency of disk operations and hardware utilization.

The extended version of this paper[8] includes details essential for a real-word implementation of CASPaxos including information on how to grow/shrink the number of nodes in the cluster and on how to efficiently remove a key/value pair not relying on the tombstones to save storage space.

[7] https://github.com/rystsov/perseus
[8] https://arxiv.org/abs/1802.07000

The possible applications of the protocol are key-value storages and stateful actor based services such as Microsoft Orleans[9]. The indirect contributions of our work is the proof of safety properties which is also applicable to the other variants of Paxos.

## References


1 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

2 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.

3 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.

4 Burkhard Englert, Chryssis Georgiou, Peter M Musial, Nicolas Nicolaou, and Alexander A Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *International Conference On Principles Of Distributed Systems*, pages 240–254. Springer, 2009.

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

6 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.

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


[9] https://dotnet.github.io/orleans