CASPaxos: Replicated State Machines without logs

Denis Rystsov
rystsov.denis@gmail.com

30 January 2018

Abstract

CASPaxos is a replicated state machine (RSM) protocol, an extension of Single Decree Paxos (Synod) protocol. Unlike Raft and Multi-Paxos, it doesn't use leader election and log replication, so it avoids associated complexity such as log compaction. Its symmetric peer-to-peer approach achieves optimal commit latency in the wide-area and doesn't cause cluster unavailability when a leader crashes.

The lightweight nature of CASPaxos creates new applications for RSM, e.g. instead of representing a key/value storage as a single RSM; it's possible to run an instance of CASPaxos per key to better fault tolerance and better performance on multi-core systems.

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

1 Introduction

Multi-Paxos[1] and Raft[6] protocols allow a collection of machines to work as a state machine tolerating failures and network issues. The protocols preserve liveness when at least $\lfloor N/2 \rfloor + 1$ of N machines are up and connected and preserve safety in the presence of arbitrary crash/recovery and loss of messages.

The problem of keeping RSM work when its nodes crash and network is falling apart is isomorphic to the problem of master-master replication of a linearizable distributed storage under the same conditions. So those protocols are widely used

in the industry as a foundation of such databases as Chubby[3], Etcd[1], Spanner[5], etc.

Despite the wide adoption, there are a lot of indications that the protocols are complex and cause availability problem when a leader crashes. Diego Ongaro and John Ousterhout write in "In Search of an Understandable Consensus Algorithm"[6]:

> 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 RSM protocols may lead to errors in implementations. Kyle Kingsbury made a comprehensive research[2] of distributed consistent databases and found violations of linearizability in almost every database he tested including MongoDB, Etcd, Consul, RethinkDB, VoltDB and CockroachDB.

The "There Is More Consensus in Egalitarian Parliaments" paper[7] 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.

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

Contributions. I present CASPaxos, a novel protocol for building RSM that avoids complexities of log-based systems.

Multi-Paxos is RSM built on top of a replicated log which treats every log entry as a command. The replicated log is modelled as an array of Synod instances. According to D. Ongaro and J. Ousterhout, 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

CASPaxos extends Synod, a write-once distributed register, into a rewritable distributed register (which is isomorphic to a RSM) instead of using it as a building block, so there is no composition and the associated complexity.

According to an experimental study[6] and the number of open source implementations[3] Raft succeeded in its goal to be understandable. However, its complexity is still comparable with Multi-Paxos: both systems[3][6] 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 its 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[7] paper: (1) optimal commit latency in the wide-area when tolerating one and two 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.

The formal proof is included into the appendix, Tobias Schottdorf[5] and Greg Rogers[6] model checked protocol with TLA+ models, and implementation was

[3] https://raft.github.io/#implementations
[4] https://github.com/gryadka/js
[5] https://tschottdorf.github.io/single-decree-paxos-tla-compare-and-swap
[6] https://medium.com/@grogepodge/tla-specification-for-gryadka-c80cd625944e

tested with faults injection methodology.

In the following sections, I describe the CASPaxos protocol, cluster membership change and evaluate a CASPaxos-based key/value storage.

2 Algorithm

We begin by briefly describing the Synod protocol from the perspective of master-master replication, followed by an overview of its extension into CASPaxos.

2.1 Synod

An implementation of the Synod protocol is a distributed register which a client can initialize only once. If several clients try to initialize a register concurrently, then the requests either prevent each other from continuing, or a single initialization succeeds. Once a client receives confirmation, all the follow-up initializations must result in a conflict and return the accepted value.

The system belongs to the CP category of the CAP theorem and keeps working without compromising safety when at least $\lfloor N/2 \rfloor + 1$ of N nodes are up and connected; with more failures, it gives up availability.

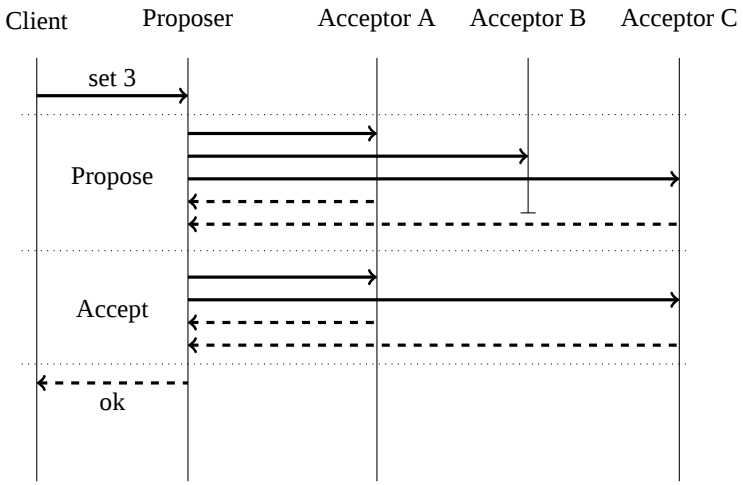
The roles of nodes in the system are:

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

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

3. **Acceptors** store the accepted value, the system should have $2F+1$ acceptor to tolerate F failures.

It's convenient to use tuples as ballot numbers, to generate it a proposer combines an increasing counter with its numerical ID: (counter, ID). To compare ballot tuples, we should compare the counter parts and use ID only as a tiebreaker. When a proposer receives a conflict from an acceptor, it should fast-forward its counter to avoid a conflict on the next request.



2.2 CASPaxos

CASPaxos is a rewritable distributed register. Clients change its state by submitting side-effect free functions which take the current state as an argument and yield new state 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 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 a value to a proposer.	A client submits a change function to a proposer.
The proposer generates a ballot number, B, and send a prepare message containing that number to the acceptors.	The proposer generates a ballot number, B, and send a prepare message containing that number to the acceptors.
An acceptor: Returns a conflict if it already saw a greater ballot number.	An acceptor: Returns a conflict if it already saw a greater ballot number.

Synod	**CASPaxos**
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.	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 the latest value and its ballot number.
The proposer waits for the $F+1$ confirmations	The proposer waits for the $F+1$ confirmations
If they all contain an empty value then the proposer writes "ok" to the result variable and sends an accept message containing the ballot number B and the proposing value to the acceptors.	The proposer defines the current state as $\emptyset$ if each confirmation contains an empty value; if at least one confirmation contains a tuple, then the proposer picks the value of the tuple with the highest ballot number as the current state.
If at least one message contains a tuple then the proposer picks the tuple with the highest ballot number, writes ("conflict", the tuple's value) to the result variable and sends an accept message with the generated ballot number B and the tuple's value to the acceptors.	Then it invokes the change function passing the current state as an argument to calculate the new state and sends it with the generated ballot number B as an accept message to the acceptors.
An acceptor: 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	An acceptor: 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
The proposer waits for the $F+1$ confirmations	The proposer waits for the $F+1$ confirmations.
The proposer returns the result variable to the client	The proposer returns the new state to the client.

As you can see, the CASPaxos's state transition is almost identical to the Synod's initialization.

If we use "$x \to$ if $x = \emptyset$ then val else x" as the change function then it becomes identical to Synod's attempt to initialize the register with the val value.

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

1. To **initialize** a register with val_0 value: "$x \to$ if $x = \emptyset$ then $(0, val_0)$ else x"
2. To **update** a register to value val_1 if the current version is 5: "$x \to$ if $x = (5, *)$ then $(6, val_1)$ else x"
3. To **read** a value: "$x \to x$"

With this specialization, the protocol becomes almost indistinguishable from Bizur[10].

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 to 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 Cluster membership change

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

1. *How to change 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 $\lfloor N/2 \rfloor$ crashes, a way to increasing/ decreasing size of a cluster is also a way to increasing/decrease resiliency of the system.

2. *How to replace permanently failed nodes.* CASPaxos tolerates transient failures but when the failures are permanent eventually more than $\lfloor N/2 \rfloor$ nodes crash, and the system becomes unavailable. A replacement of a failed node in the N nodes cluster can be modelled as a shrinkage to $N-1$ followed by expansion to N.

The process of memberschange is based on Raft's idea of joint consensus[6] where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

The proof of this idea with application to CASPaxos is based on two observation:

1. **Flexible quorums** It has been observed before that in a Paxos based system the only requirement for the "prepare" and "accept" quorums is intersection [2][4][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. The same result applies to CASPaxos, see the proof in appendix B.

2. **Filter equivalence** If a change in the behaviour of the CASPaxos cluster can be explained by delaying or omitting the messages between the nodes of the cluster then the change doesn't affect consistency since it tolerates the interventions of such kind. It gives freedom in changing the system as long as the change can be modelled as a message filter on top of the unmodified system.

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.

3. Pick any proposer and execute the identity change 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 filter equivalence principle, so the system keeps being correct.

After it finishes the system also works as a $2F + 2$ nodes cluster with $F + 1$ "prepare" quorum and $F + 2$ "accept" quorum.

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

The protocol for changing the set of acceptors from $A_1 \cdots A_{2F+2}$ to $A_1 \cdots A_{2F+3}$ is simpler 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.

To reduce the size of the cluster, the same steps should be executed in the reverse order.

How many crashes can we tolerate if the crashes occur while the reconfiguration is in progress? The reconfiguration procedure is a sequence of atomic changes, at each step a system is in a correct state either from the perspective of previous or next configuration. So it can tolerate the number of failures the configuration allows. For example, during $A_1 \cdots A_{2F+1}$ to $A_1 \cdots A_{2F+2}$ reconfiguration the system can tolerate F failures. On the $A_1 \cdots A_{2F+2}$ to $A_1 \cdots A_{2F+3}$ path the fault tolerance increases from F to $F + 1$.

2.2.3 Low-latency and high-throughput consensus across WAN deployments

WPaxos[9] 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 being an extension of Synod provides flexible quorums capabilities it can benefit from the same idea too.

3 A CASPaxos-based key/value storage

The lightweight nature of CASPaxos opens new simple ways for designing distributed systems with complex behaviour. 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 acknoledges[6] that EPaxos[7] can achieve higher performance than Raft by using 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 lightweight nature of CASPaxos and run a RSM per key to achieving uniform load balancing across all replicas (thus higher throughput).

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

3.1 How to delete a record

The proposed variant of CASPaxos supports only update (change) operation so to delete a value a client should update a register with an empty value. 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: a promise and an associated ballot number.

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

	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 the information associated with the register and a request hits the system during the removal 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. A multi-step removal process can be used to avoid the availability impact.

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

1. On a delete request a system writes an empty value with regular $F+1$ accept-quorum, schedules a garbage collection operation and confirms the request to a client.

2. The garbage collection operation:

   (a) Replcates an empty value to all nodes by reading it with $2F+1$ quorum size.

   (b) Removes the empty register from every acceptor.

3.2 Low latency

The following behaviour helps CASPaxos achieve low latency:

- CASPaxos isn't leader based so a proposer should not forward all to a specific node to start executing.
- Requests affecting different key/value pairs do not interfere.
- It uses 1RTT in case most of 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.
- Ability to use user-defined functions as state transition functions reduces two steps transition process (read, modify, write) into one step process.

Some of this behaviour matches Multi-Paxos and Raft, but some doesn't. The EPaxos paper emphasizes the following latency issue of 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 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 nodes configuration deployed over wide area network in the Azure's[8] datacenters in the "West US 2", "West Central US" and "Southeast Asia" regions running on the DS4_V2 machines with SSD drive.

Each node has a colocated client which in one thread in a loop was reading, incrementing and writing back a value. 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	530 ms	680 ms	46 ms
West Central US	586 ms	726 ms	46 ms
Southeast Asia	366 ms	341 ms	345 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	23.7 ms
West US 2	Southeast Asia	171.4 ms
West Central US	Southeast Asia	191.5 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 one round trip to forward a request to the leader and to receive a responce (171.4 ms), the leader needs to write the change to the majority of nodes and get confirmations (at least 171.4 ms). Since the iteration consists of reading and writing in total "West US 2" node requires at least $685.6\text{ms} = 2 \cdot (171.4\text{ms} + 171.4\text{ms})$. For the "West Central US" node the estimated latecy is $725.8\text{ms} = 2 \cdot (191.5\text{ms} + 171.4\text{ms})$, for "Southeast Asia" it's $342.8\text{ms} = 2 \cdot 171.4\text{ms}$.

Gryadka doesn't forward requests so the corresponding estimated latencies are $47.4\text{ms} = 2 \cdot 23.7\text{ms}$, $47.4\text{ms} = 2 \cdot 23.7\text{ms}$ and $342.8\text{ms} = 2 \cdot 171.4\text{ms}$.

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

Network fluctuations may explain the minor difference between estimated and measured latencies.

As you see, the difference in the underlying consensus protocol yields a significant difference in the performance of the system.

3.3 Fault-tolerance

The EPaxos paper explains how 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 behaviour because all of its nodes are homogeneous and isolation of one of them doesn't affect processes running against the others. An experimental study[9] of distributed consistent databases with default settings during leader isolation supports this a priori reasoning:

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

4 Comparison with Related Work

Raft 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

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

like leader election and log replication but rearranges them differently with the focus on understandability. CASPaxos chooses a different foundation and has less moving parts. As a result, its implementation is $\frac{1}{4}$ of Raft's regarding lines of code. The symmetric peer-to-peer approach of CASPaxos allows it to experience isolation of a node without impacting clients, on the contrary, an isolation of a leader in Raft makes the whole cluster unavailable until a new leader isn't elected.

Bizur 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. The Bizur paper doesn't specify how to remove values (buckets) other than by creating tombstones which may lead to space inefficiency.

EPaxos is a leaderless variant of Multi-Paxos which allows concurrent execution of non-interfering commands, CASPaxos is also a leaderless protocol, but it doesn't allow concurrent state transition. However, in some cases, it provides comparable functionality with simpler design. For example, key-value storage with independent key operations can be modelled as a single EPaxos-based RSM or as a system with an instance of CASPaxos 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 are complex and have latency issues during failures, they find wide adoption in the industry as a foundation of distributed databases. However, when looking closely at the applications, many of them are used to implement 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 better resiliency guarantees.

The protocol has the formal proof, was model checked with TLA+ and implementation was tested using fault-injection methodology.

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 (a proposer received at least $F+1$ confirmations of an "accept" message) 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 changes changing one state into another. Let's formalize it. We start by defining this relation between the successfully accepted messages (denoted as $\ddot{E}^2$) and then expend it to $\bar{E}^2$.

By definition of a proposer 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 events which correspond to the confirmations of an accept message. By definition the following properties are true:

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

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

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

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 acknowledgement or confirmation depending on type x.

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 proofs the theorem 1. □

Before proving the 5 let's define $\ddot{E}^1$ as a set of promised events (prepare phase) and $x.r$ as for $x \in \ddot{E}^2$ as events which correspond to the confirmations of an prepare message. 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 $x.w$ intersects with $y.r$. Let $n \in N$, and $w \equiv x.w|_n$ and $u \equiv y.r|_n$ are accepted and promised events on node n. We know that event w happened before u in n's timeframe because an acceptor doesn't accept messages with lesser ballot numbers than they already saw and $w.b = x.b < y.b = u.b$ implies

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

Let "promise" events P have the following structure $\{\text{ts}, \text{b}, \text{ret} : \{\text{b}, \text{s}\}\}$ where ts - local time, b - promised ballot number, ret.b - a ballot number of the accepted state and ret.s is the state itself. "accept" events A's structure is $\{\text{ts}, \text{b}, \text{s}\}$.

By definition "promise" confirmation returns the latest state, let's write it formally

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

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:

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

By definition we pick a state with max ballot number out of quorum of promise confirmations as the current state 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 $x.b \leq I^{-1}(y).b$.

□

B FPaxos

In the proof of CASPaxos A we didn't use the size of the promise/accept quorums and only used their intersection, so the same proof proves FPaxos observation too.

References

[1] Leslie Lamport, *"Paxos Made Simple"*. 2001.

[2] Butler W. Lampson *"The ABCDs of Paxos"*. 2001.

[3] Tushar Chandra, Robert Griesemer, Joshua Redstone *"Paxos Made Live - An Engineering Perspective"*. 2007.

[4] Leslie Lamport, Dahlia Malkhi, Lidong Zhou *"Vertical Paxos and Primary-Backup Replication"* 2009.

[5] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., at al. *"Spanner: Googles globally distributed database"*. 2012.

[6] Diego Ongaro, John Ousterhout *"In Search of an Understandable Consensus Algorithm"*. 2013.

[7] Iulian Moraru, David G. Andersen, Michael Kaminsky *"There Is More Consensus in Egalitarian Parliaments"*. 2013.

[8] Heidi Howard, Dahlia Malkhi, Alexander Spiegelman *"Flexible Paxos: Quorum intersection revisited"*. 2016.

[9] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Tevfik Kosar *"WPaxos: Ruling the Archipelago with Fast Consensus"*. 2017.

[10] Ezra N. Hoch, Yaniv Ben-Yehuda, Noam Lewis, Avi Vigder *"Bizur: A Key-value Consensus Algorithm for Scalable File-systems"*. 2017.