On the correctness of Egalitarian Paxos

Pierre Sutra

Version of Record: https://www.sciencedirect.com/science/article/pii/S002001901930184X
Manuscript_330c208c4de86a1c4d13ba26105ce35b

On the correctness of Egalitarian Paxos

Pierre Sutra

Telecom SudParis
9, rue Charles Fourier
91000 Évry, France

Abstract

This paper identifies a problem in both the TLA$^+$ specification and the implementation of the Egalitarian Paxos protocol. It is related to how replicas switch from one ballot to another when computing the dependencies of a command. The problem may lead replicas to diverge and break the linearizability of the replicated service.

Keywords: Distributed systems; Fault tolerance; State-machine replication; Consensus.

1. Introduction

State-machine replication (SMR) is a fundamental technique to build dependable services. With SMR, a service is replicated across a set of distributed processes. Replicas apply to their local copies the commands that access the replicated service following a total order. This order is implemented using a repeated agreement protocol, or *consensus*, on the next command to execute.

Some recent works [1, 2] observe that it is not necessary to build a total order over the commands submitted to the service. To maintain service consistency and provide the illusion of having no data replication, ordering non-commuting commands suffices. Egalitarian Paxos (EPaxos) [3] is a novel protocol built upon this insight. To construct the partial order, EPaxos replicas agree on the dependencies of each command submitted to the service. A replica executes commands according to the graph formed by these dependencies, that is, if c depends on d, then c is executed after d.

To agree on the dependencies of a command, EPaxos follows a common pattern: replicas successively join asynchronous ballots, and during a ballot, they try to make a decision.

Surprisingly, the TLA$^+$ specification and the Golang implementation of EPaxos use a single variable at each replica to track progress across ballots. This paper shows that this is not sufficient. We exhibit an admissible execution in which replicas disagree on the dependencies of a command, breaking consistency.

Outline. Section 2 recalls the traditional schema of repeated asynchronous ballots to reach consensus. Section 3 gives an overview of the EPaxos algorithm. Section 4 depicts our counter-example. Section 5 closes this paper. An appendix follows that contains the TLA$^+$ specification of the counter-example. The specification is also available online [4].

2. Solving consensus

The classical approach to solve consensus is to execute a sequence of asynchronous rounds, or *ballots*. Each ballot is identified with a natural, its *ballot number*. As usual, we refer to a ballot with its ballot number and assume that processes starts the agreement at ballot 0.

During a ballot, a *quorum* of processes attempt to agree on some proposed value. To this end, each ballot is split into three distinct phases. Before it participates in a ballot, a process first *joins* it (the *prepare* phase). Then, it may *vote* for some proposed value (*accept* phase). A value is *chosen* at a ballot when a quorum of processes voted for it. A process *decides* some value once it knows that this value was chosen at some ballot (*learn* phase).

The size of a quorum depends on the time taken by the protocol to reach a decision in a ballot. Consider a system of $n = 2f + 1$ processes of which at most f may fail-stop. In Paxos [5], any majority of the processes (that is, at least $f + 1$ of them) is a quorum at every ballot. Differently, Fast Paxos [6] distinguishes fast and classic ballots. Whereas a majority of processes is a quorum for a classic ballot, a quorum of a fast ballot contains at least $f + \lfloor \frac{f+1}{2} \rfloor + 1$ processes.

Reaching consensus requires to choose a unique value among the proposals. To this end, the following invariant is maintained across ballots during an execution:

> (INV) if a value v is chosen at some ballot b, then for every ballot $b' \geq b$, if u is chosen at b', $u = v$ holds.

To build this invariant, consensus algorithms commonly rely on the following assumptions.

(A1) A process can join a ballot b only if it did not join some ballot $b' > b$ previously.

(A2) A process may only cast a vote for the last ballot it has joined.

(A3) The vote of a process at a ballot is irrevocable. This means that if a process votes for some value u at ballot b, it cannot later vote at ballot b for some other value $v \neq u$.

(A4) The votes of any two processes at a ballot are identical, that is, if p and q vote at ballot b for respectively u and v, then $u = v$.

(A5) Consider two quorums Q and Q' defined respectively at ballot b and b'. Then, the intersection of Q and Q' is non-empty.

To maintain invariant (INV), the consensus protocol tracks the values chosen at prior ballots when it progresses to a new ballot. In detail, a process p that aims at advancing to ballot b, first seals prior ballots. To this end, for every ballot $b' < b$, and every quorum Q at b', p asks a process $q \in Q$ to join b. By invariants (A1) and (A2), sealing prevents new value to be chosen at a lower ballot than b.

Once a quorum Q of processes have joined ballot b, p computes the values chosen prior to b. Assume that some value v is chosen at ballot $b-1$ by a quorum Q'. By invariants (A3-A5), v is unique, and some process $q \in Q \cap Q'$ voted for v at ballot $b-1$. Hence, process p discovers v when it inquiries the processes in Q at the time they join ballot b.

Now if v does not exist, by induction, p must propose what was voted at ballot $b-2$, etc. In other words, p proposes at ballot b the value for which some process in Q voted at the largest ballot prior to b. If no such value exists, p is free to pick any value as its proposal. If (INV) is true up to ballot $b-1$, then the invariant is maintained at ballot b.

The proposal of p is a so-called *safe* value. It maintains invariant (INV) at ballot b by over-approximating what was chosen at prior ballots. In the literature [2], the construction of a safe value makes use of three variables at each process. The first one, denoted hereafter *bal*, is the last ballot joined so far by a process. The second variable *vbal* is the last ballot at which the process voted for some value. Variable *vval* contains the value that was voted at ballot *vbal*.

The TLA$^+$ specification of EPaxos and its implementation rely on not two ballot variables, as mentioned above, but a single one to reach an agreement on the dependencies of a command. In the remainder of this paper, we explain that this mechanism is flawed. To this end, we give an overview of the EPaxos protocol, then we exhibit an admissible run that breaks safety.

3. Overview of EPaxos

Egalitarian Paxos (EPaxos) is a recent protocol to implement state-machine replication and construct fault-tolerant distributed services. EPaxos is leaderless, and it orders commands in a decentralized way, without relying on a distinguished process. As in [1, 2], EPaxos exploits the commutativity between commands submitted to the replicated service to improve its performance In the most favorable case, that is when there is no concurrent non-commuting command, the protocol decides the next command to execute after one round-trip to the closest fast quorum.

EPaxos maintains the consistency of the replicated service, namely its linearizability [7], with the help of three mechanisms. First, processes running the protocol agree on a partially ordered set of commands, or *execution graph*. They apply the commands following some linearization of this graph. Second, any two non-commuting commands must be ordered in the graph. Last, the execution graph grows monotonically at each process. This means that if command c but not d is in the execution graph at time t, then d cannot precede c at a later time $t' > t$.

To build the execution graph, for each command c submitted to the system, EPaxos constructs a set of dependencies $dep(c)$. Command c is *committed* at a process p once $dep(c)$ is known at p. The dependencies of c precede it in the execution graph built at p. The EPaxos protocol ensures two core invariants:

(E1) Any two processes agree on the dependencies of a command.

(E2) For any two non-commuting committed commands c and d, $c \in dep(d)$ or the converse holds.

The conjunction of these two invariants ensure that processes execute committed non-commuting commands in the same order.

To agree on the dependencies of a command c, EPaxos employs a variation of Fast Paxos. The first (so-called) *pre-accept* phase of the agreement is not coordinated. To propose a command c, a process p propagates c to a quorum. Upon receiving command c, a process sends all the commands conflicting with c (that it is aware of) back to p. If all the processes return the same dependencies, a spontaneous agreement occurs. Otherwise, a classical voting phase (aka., accept phase) takes place.

Let us notice that if ballot b is not coordinated, processes may return different values for $dep(c)$, and invariant A4 would not hold anymore. When b is fast, (A4) is replaced with the following Fast Paxos invariant [6]:

(A4') For any two quorums Q and Q' at ballot b, and any quorum Q'' at some ballot b', $Q \cap Q' \cap Q'' \neq \varnothing$.

Invariant (A4') ensures that a process starting ballot b' (a *recovery* in EPaxos parlance) observes at most a single chosen value at ballot b, the one with the most votes.

Differently from Fast Paxos, only ballot 0 is fast in EPaxos and allows a spontaneous agreement to form.[1] In addition, when a process proposes a command, it includes its local computation of $dep(c)$ in its proposal. This allows to reduce the size of a fast quorum at ballot 0 to $f + \lfloor \frac{f+1}{2} \rfloor$ processes.

4. Breaking safety

As detailed in Section 2, consensus algorithms commonly employ two ballot variables at a process. These variables track the last joined ballot (*bal*) and the last ballot at which a value was voted (*vbal*).

However, both the TLA$^+$ specification of EPaxos [8, pages 109–123] and its implementation [9] use only *bal*.[2] When a process receives a message to join a ballot $b > bal$, it sends back *bal* and *vval*, then updates *bal* to b. This is surprising as the pseudo-code [3, Figure 3] seems to correctly identify the

[1] During a recovery, the TryPreAccept messages serve solely to ensure invariant (E2).

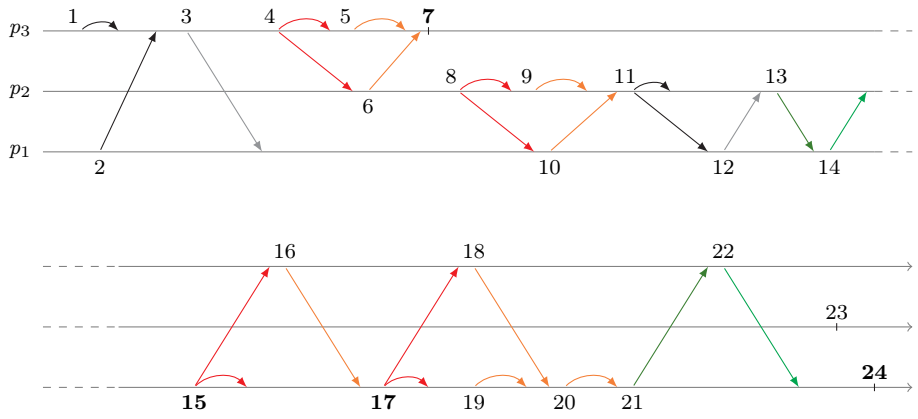
[2] The specification and the implementation of EPaxos call *ballot* this unique variable.

Step	Process	$(status, bal, dep)$ for $\langle p_1, 1\rangle$
7	p_3	"accepted", 1, $\{c_2\}$
15	p_1	"accepted", 2, $\{\}$
	p_2	"accepted", 2, $\{\}$
	p_3	"accepted", 1, $\{c_2\}$
17	p_1	"accepted", 2, $\{\}$
	p_2	"accepted", 2, $\{\}$
	p_3	"accepted", 3, $\{c_2\}$
24	p_1	"accepted", 2, $\{\}$
	p_2	"committed", 2, $\{\}$
	p_3	"committed", 4, $\{c_2\}$

(a) Key states

Color	Type
→	"pre-accept"
→	"pre-accept-reply"
→	"prepare"
→	"prepare-reply"
→	"accept"
→	"accept-reply"

(b) Message labels



(c) Execution timeline

Figure 1: Breaking safety in EPaxos

pattern, mentioning to send back "the most recent ballot number accepted" in the consensus instance.

Avoiding the use of a second ballot variable is not possible. Figure 1 illustrates how to break safety by executing a well-chosen sequence of steps.

In this figure, the system consists of three processes $\{p_1, p_2, p_3\}$. Processes p_1 and p_3 propose respectively to compute the dependencies of commands c_1 and c_2. This computation occurs respectively in the consensus instances $\langle p_3, 1\rangle$ and $\langle p_1, 1\rangle$. At the end of the execution, the value of $dep(c_2)$ is $\varnothing$ at process p_2, while it equals $\{c_1\}$ at processes p_1.

Appendix A provides the TLA$^+$ code of the execution given in Figure 1. This counter-example is located in a set of states which is too large to explore in a reasonable amount of time with the TLA$^+$ model checker. As a consequence,

the specification in Appendix A directly injects the admissible execution with the help of a history variable. The model checker can be then used to validate that the counter-example is actually feasible. The history variable introduced in Appendix A is a counter. Its value coincides with the numbering of the steps in Figure 1.

The disagreement depicted in Figure 1 is obtained by executing consecutive recovery phases. It is based on the following observation: if *vbal* is not used, a process that accepted a value u at a ballot b, then later joins $b' > b$ is in a state identical to having accepted u at ballot b'.

In detail, the execution at Figure 1 consists of the following steps:

(1-2) Process p_3 proposes command c_1, while p_1 proposes command c_2.

(3) p_3 returns a pre-accept message with $dep(c_1) = \{c_2\}$.

(4-7) p_3 partially recovers by contacting the quorum $\{p_3, p_2\}$. This leads to the fact that $dep(c_1) = \{c_2\}$ is accepted at ballot 2 (step 7).

(8-14) p_2 executes a full recovery using $\{p_1, p_2\}$. Before step 15, $dep(c_1) = \{\}$ is accepted at process p_2.

(15-17) p_1 executes a partial recovery with quorum $\{p_1, p_3\}$. The state of instance $\langle p_1, 1\rangle$ at process p_3 is now ("accepted", $3, \{c_2\}$).

(21-22) p_1 executes a full recovery using the same quorum. As p_3 holds the highest ballot, its value for $dep(c_2)$ is accepted at ballot 4.

(23-24) Processes p_2 and p_1 decide respectively $\varnothing$ and $\{c_1\}$ for $dep(c_2)$.

5. Conclusion

Egalitarian Paxos (EPaxos) is a recent protocol to implement state-machine replication and construct fault-tolerant distributed services. In the common case, EPaxos delivers a command after one round-trip to the closest fast quorum. On the contrary to prior works, such as Generalized Paxos [2], a leader does not need to solve conflicts between non-commuting commands. These two properties make the protocol particularly appealing for geo-distributed systems.

The repeated consensus procedure of EPaxos must rely on two ballot variables to track the progress at each replica. This paper shows that if this is not the case, as in the TLA$^+$ specification and the Golang implementation, breaking safety is possible.

Acknowledgments

This research is partly supported by the RainbowFS project of Agence Nationale de la Recherche, France, number ANR-16-CE25-0013-01a.

References

[1] F. Pedone, A. Schiper, Handling message semantics with Generic Broadcast protocols, Distributed Computing 15 (2002) 97–107.

[2] L. Lamport, Generalized Consensus and Paxos, Tech. Rep. MSR-TR-2005-33, Microsoft (March 2005).

[3] I. Moraru, D. G. Andersen, M. Kaminsky, There is More Consensus in Egalitarian Parliaments, in: ACM Symposium on Operating Systems Principles (SOSP), 2013, pp. 358–372.

[4] On the correctness of Egalitarian Paxos, `https://github.com/otrack/on-epaxos-correctness`, online; accessed January 2019.

[5] L. Lamport, The part-time parliament, ACM Transactions on Computer Systems 16 (2) (1998) 133–169.

[6] L. Lamport, Fast Paxos, Distributed Computing 19 (2) (2006) 79–103.

[7] M. Herlihy, J. Wing, Linearizability: a Correcteness Condition for Concurrent Objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.

[8] I. Moraru, Egalitarian Distributed Consensus, Ph.D. thesis, School of Computer Sciences, Carnegie Mellon University (August 2014).

[9] EPaxos, `https://github.com/efficient/epaxos`, online; accessed January 2019.

Appendix A. The counter example

Appendix A.1. TLA$^+$ specification

MODULE *CounterExample*

EXTENDS *EgalitarianPaxos*, *TLC*

CONSTANTS $p1$, $p2$, $p3$, $c1$, $c2$
VARIABLES *HIndex*

$MCReplicas \triangleq \{p1, p2, p3\}$
$MCCommands \triangleq \{c1, c2\}$
$MCFastQuorums(X) \triangleq$ IF $X = p1$ THEN $\{\{p1, p3\}\}$
ELSE IF $X = p2$ THEN $\{\{p1, p2\}\}$
ELSE $\{\{p2, p3\}\}$
$MCSlowQuorums(X) \triangleq MCFastQuorums(X)$
$MCMaxBallot \triangleq 5$

$AdvanceHistory(pos) \triangleq \land HIndex = pos$
$\land HIndex' = HIndex + 1$

$NewInit \triangleq$
$\land HIndex = 1$
$\land sentMsg = \{\}$
$\land cmdLog = [r \in Replicas \mapsto \{\}]$
$\land proposed = \{\}$
$\land executed = [r \in Replicas \mapsto \{\}]$
$\land crtInst = [r \in Replicas \mapsto 1]$
$\land leaderOfInst = [r \in Replicas \mapsto \{\}]$
$\land committed = [i \in Instances \mapsto \{\}]$
$\land ballots = 1$
$\land preparing = [r \in Replicas \mapsto \{\}]$

$NewNext \triangleq \lor (AdvanceHistory(1) \land Propose(c1, p3))$
$\lor (AdvanceHistory(2) \land Propose(c2, p1))$
$\lor (AdvanceHistory(3) \land Phase1Reply(p3))$
$\lor (AdvanceHistory(4) \land SendPrepare(p3, \langle p1, 1\rangle, \{p2, p3\}))$
$\lor (AdvanceHistory(5) \land ReplyPrepare(p2))$
$\lor (AdvanceHistory(6) \land ReplyPrepare(p3))$
$\lor (AdvanceHistory(7) \land PrepareFinalize(p3, \langle p1, 1\rangle, \{p2, p3\}))$
$\lor (AdvanceHistory(8) \land SendPrepare(p2, \langle p1, 1\rangle, \{p1, p2\}))$
$\lor (AdvanceHistory(9) \land ReplyPrepare(p1))$
$\lor (AdvanceHistory(10) \land ReplyPrepare(p2))$
$\lor (AdvanceHistory(11) \land PrepareFinalize(p2, \langle p1, 1\rangle, \{p1, p2\}))$
$\lor (AdvanceHistory(12) \land Phase1Reply(p1))$
$\lor (AdvanceHistory(13) \land Phase1Slow(p2, \langle p1, 1\rangle, \{p1, p2\}))$

$\lor$ ($AdvanceHistory(14) \land Phase2Reply(p1)$)
$\lor$ ($AdvanceHistory(15) \land SendPrepare(p1, \langle p1, 1\rangle, \{p1, p3\})$)
$\lor$ ($AdvanceHistory(16) \land ReplyPrepare(p3)$)
$\lor$ ($AdvanceHistory(17) \land SendPrepare(p1, \langle p1, 1\rangle, \{p1, p3\})$)
$\lor$ ($AdvanceHistory(18) \land ReplyPrepare(p3)$)
$\lor$ ($AdvanceHistory(19) \land ReplyPrepare(p1)$)
$\lor$ ($AdvanceHistory(20) \land ReplyPrepare(p1)$) answer both ballots
$\lor$ ($AdvanceHistory(21) \land PrepareFinalize(p1, \langle p1, 1\rangle, \{p1, p3\})$)
$\lor$ ($AdvanceHistory(22) \land Phase2Reply(p3)$)
$\lor$ ($AdvanceHistory(23) \land Phase2Finalize(p2, \langle p1, 1\rangle, \{p1, p2\})$)
$\lor$ ($AdvanceHistory(24) \land Phase2Finalize(p1, \langle p1, 1\rangle, \{p1, p3\})$)

Appendix A.2. Model

```
CONSTANTS
p1 = p1
p2 = p2
p3 = p3
c1 = c1
c2 = c2
none = none
Replicas <− MCReplicas
Commands <− MCCommands
FastQuorums <− MCFastQuorums
SlowQuorums <− MCSlowQuorums
MaxBallot <− MCMaxBallot
Init <− NewInit
Next <− NewNext
SPECIFICATION Spec
PROPERTY Consistency
```