EPaxos Dependency Set Compaction Bug

September 12, 2021

In this document, we outline a minor bug in an optimization that EPaxos performs to keep dependency sets small [2, 1].

1 EPaxos Dependency Set Compaction

When an EPaxos replica R receives a command x from a client, it first assigns the command to an instance $R.i$. It then computes the command's dependencies: the set of instances $Q.j$ that contain a command that conflict with x. R forwards these dependencies to other replicas, which also compute the command's dependencies.

Over time, as more and more commands are executed, these sets of dependencies grow larger and larger. Every command starts to depend on more and more commands. The EPaxos paper proposes the following mechanism to keep these dependency lists small (see Section 4.5):

> Instead of including all interfering instances, we include only N dependencies in each list: the instance number $R.i$ with the highest i for which the current replica has seen an interfering command (not necessarily committed). If interference is transitive (usually the case in practice) the most recent interfering command suffices, because its dependency graph will contain all interfering instances $R.j$, with $j < i$.

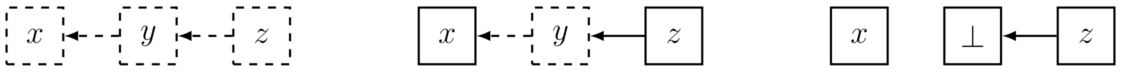
There are some details omitted from this description. When a replica receives a pre-accept request from a leader, how does it merge its locally computed dependencies with the dependencies from the leader? And when a leader takes the slow path, how does it merge dependencies? The TLA+ specification does not implement dependency set compaction, so we can't look there for clarification.

The Go implementation seems to perform all merging using an element wise maximum. More specifically, we represent a dependency set for an EPaxos deployment with n replicas as a list $d = i_1, \ldots, i_n$ where entry i_m is the highest i for which replica m has seen an interfering command. To merge two dependency sets $i_1, \ldots, i_n$ and $j_1, \ldots, j_n$, we compute an element wise maximum: $\max(i_1, j_1), \ldots, \max(i_n, j_n)$. Throughout the rest of the paper, we assume that EPaxos uses this mechanism to merge dependency sets.

2 The Bug

We now present an execution of EPaxos in which two replicas execute conflicting commands in different orders. The gist of the bug is that we create a sequence of commands x, y, z that all conflict with one another. We get y to depend on x and then z to depend on y (and not x). We get x and z chosen but then recover y's instance, changing it to a noop. z had and transitive dependence on x, but this transitive dependence is erased when we recover y's instance with a noop. This allows replicas to execute x and z in either order. The bug is summarized in Figure 1.

Now, we outline the execution in detail. **pre** is short for pre-accepted, **acc** is short for accepted, and **comm** is short for committed. We do not show sequence numbers or ballot


(a) Initially, commands x, y, and z are all uncommitted. y depends on x and z depends on y but not x. z conflicts with x, so without any optimizations, it would have an edge to x. However, EPaxos' dependency compaction optimization allows to omit the edge from z to x because of the transitive dependence through y.

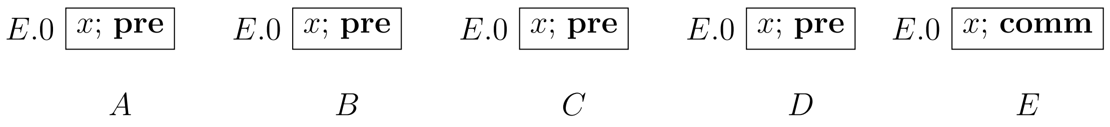
(b) Next, x and z are both committed, but y remains uncommitted for some time.

(c) Eventually, a replica recovers y's instance and gets a noop, $\perp$, chosen. $\perp$ doesn't conflict with any other commands so is committed without any dependencies. x and z conflict but do not have any path between them. This allows some replicas to execute x before z and others to execute z before x.

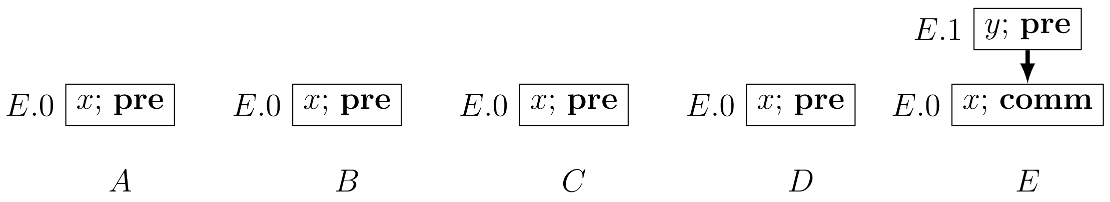
Figure 1: An cartoon overview of the bug.

numbers because they aren't relevant to the bug. Commands x, y, and z are all assumed to conflict. We use an EPaxos deployment with 5 replicas: A, B, C, D, and E. At each step of the execution, we draw the command log at each replica.

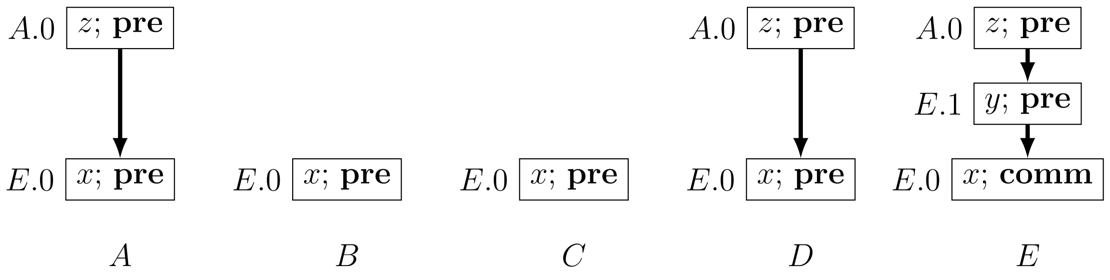
Initially, replica E receives command x. E chooses x on the fast path and sends out commit messages to the other replicas. These commit messages are delayed.



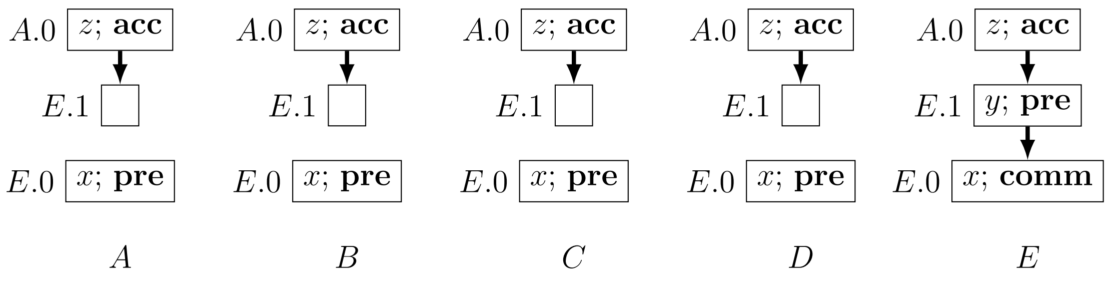
Then, replica E receives command y and pre-accepts it locally. It sends out pre-accept messages to the other replicas, but they are lost.



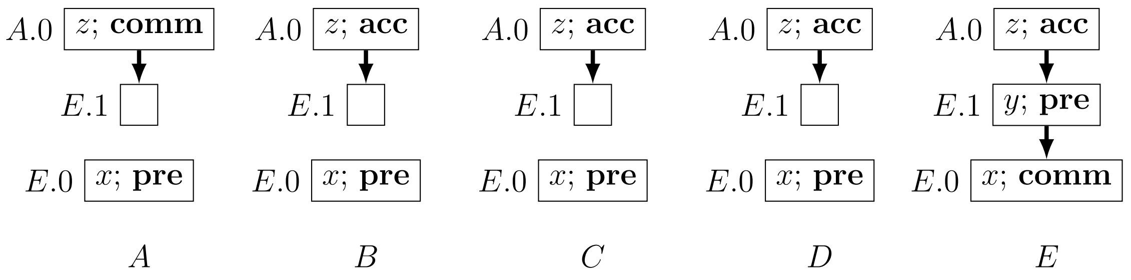
Then, replica A receives command z and pre-accepts it locally. It sends out pre-accept messages to the other replicas. Replicas D and E receive the pre-accept messages and process them. Note that replica E computes z's dependencies to be $\{E.1\}$. E does not include $E.0$ in the dependencies of z.



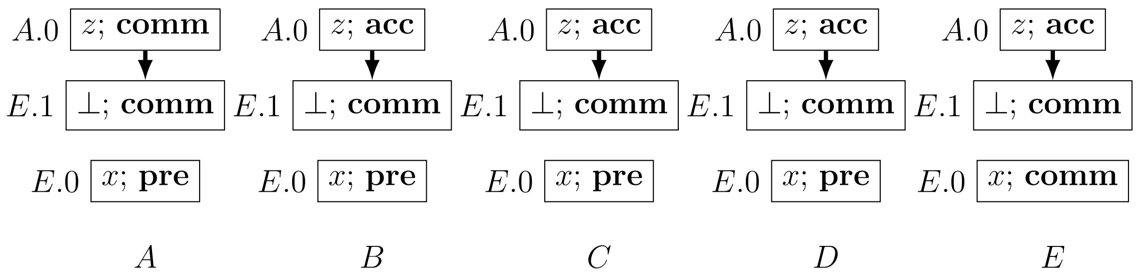
Replica A receives pre-accept replies from replica D and E. After a timeout, it takes the slow path. It computes z's dependencies to be $\{E.1\}$. Again note that z does not include $E.0$ in its dependencies. A accepts z locally and sends out accept messages to all other replicas. All other replicas receive the accept message and send back a reply.



Replica A receives accept replies from all other replicas and commits z locally. It sends commit messages to all other replicas, but the messages are delayed.



After a timeout, replica A notices that $E.1$ has not been chosen. This prevents z from being executed, so replica A begins to recover instance $E.1$. It sends prepare requests to replicas A (itself), B, and C. All three replicas reply without having seen $E.1$, so A begins the slow path to get a noop chosen (see line 37 of Figure 3 in [2]). We denote a noop with $\perp$. Talking exclusively to replicas A, B, and C, replica A eventually gets the $\perp$ chosen and broadcasts a commit message to all replicas. All replicas receive the commit message. Note that a $\perp$ has no dependencies.



Replica B receives the commit message for $E.0$ from E and executes x. It then receives the commit message for $A.0$ from A and executes z. Replica C receives the messages in the opposite order and executes z and then x. The two replicas execute conflicting commands in different orders, which is a bug.

References

[1] Iulian Moraru, David G Andersen, and Michael Kaminsky. A proof of correctness for egalitarian paxos. Technical report, Technical report, Parallel Data Laboratory, Carnegie Mellon University, 2013.

[2] 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.